UNIVERSITE DE NANTES

ÉCOLE DOCTORALE

« SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATHEMATIQUES »

Année : 2010

**Thèse de Doctorat de l'Université de Nantes**

Spécialité : AUTOMATIQUE ET INFORMATIQUE APPLIQUÉE

Présentée et soutenue publiquement par :

**Rola KASSEM**

le 17 Juin 2010
à l'IRCCyN, Nantes

**TITRE**

# Langage de description d'architecture matérielle pour les systèmes temps réel

**Jury**

| | | |
|---|---|---|
| Président : | Isabelle PUAUT, | Professeur à l'Université de Rennes, IRISA/INRIA, Rennes |
| Rapporteurs : | Pascal SAINRAT, | Professeur à l'Université Paul Sabatier, IRIT, Toulouse |
| | Michel AUGUIN, | Directeur de Recherche CNRS, LEAT, Valbonne |
| Examinateurs : | Isabelle PUAUT, | Professeur à l'Université de Rennes, IRISA/INRIA, Rennes |
| | Yvon TRINQUET, | Professeur à l'Université de Nantes, IRCCyN, Nantes |
| | Jean-Luc BÉCHENNEC, | Chercheur CNRS, IRCCyN, Nantes |
| | Mikaël BRIDAY, | Maître de Conférences, IRCCyN, Nantes |

**Directeur de thèse : Yvon TRINQUET**
Laboratoire : IRCCyN, Nantes
**Encadrant de thèse : Jean-Luc BÉCHENNEC**
Laboratoire : IRCCyN, Nantes
**Encadrant de thèse : Mikaël BRIDAY**
Laboratoire : IRCCyN, Nantes

**N° ED 503-089**

# Remerciements

Cette thèse touche à sa fin. À cette occasion, j'en profite pour remercier tous les personnes qui m'ont aidé afin de bien mener ce travail qui s'est déroulé au sein de l'équipe *Systèmes Temps Réel* de l'IRCCyN.

Je tiens à remercier particulièrement mon directeur de thèse Yvon TRINQUET, pour ses remarques judicieuses, son soutien, sa gentillesse, sa grande disponibilité et ses qualités humaines, ainsi que mes deux encadrants Jean-Luc BÉCHENNEC et Mikaël BRIDAY pour leur encadrement, leur disponibilité, les discussions intéressantes que nous avons pu avoir sur des sujets assez variés, leurs qualités professionnelles et humaines.

J'exprime toute ma gratitude envers M. Michel AUGUIN et M. Pascal SAINRAT d'avoir accepté la tâche fastidieuse de rapporter ma thèse tout comme Mme Isabelle PUAUT qui m'a fait l'honneur de participer au jury.

Tout mes remerciements vont également à Pierre MOLINARO pour sa grande disponibilité et pour ses réponses claires et instructives surtout en ce qui concerne *Galgas*. Je remercie tous l'équipe *Systèmes Temps Réel* et toutes les personnes de l'IRCCyN, Jonathan, Marwa, Émilie, Ayan, Pedro, Adrien, Di, Carlos, Charlotte, Louis-Marie, Isabelle, Nayim, Hui, Didier, ... pour les moments agréables que nous avons pu passer ensemble. Merci à tous mes amis et à toutes les personnes que j'ai côtoyés durant cette thèse et qui peut-être n'ont pas été mentionnées ici.

Enfin, j'exprime ma reconnaissance profonde à ma famille au Liban et à Biélorussie pour leur grand soutien.

Nantes, le 13 Juillet 2010
Rola KASSEM

# Résumé

Avec le développement des terminaux mobiles, les systèmes embarqués sont de plus en plus utilisés. Avec les progrès rapides de la technologie, l'architecture des composants programmables constituant ces systèmes, processeurs, co-processeurs, mémoires, périphériques, devient de plus en plus complexe. À l'opposé, la durée de vie de ces systèmes ainsi que le temps de mise en vente sur le marché ne cesse de diminuer. Par conséquent, nous constatons l'augmentation des besoins d'automatisation de la conception de ces composants programmables de plus en plus sophistiqués et complexes.

La simulation du matériel est un élément important dans la conception des systèmes embarqués. Un simulateur peut être écrit à la main (par exemple en utilisant le langage *C++*) mais cette tâche se révèle complexe, lente et sujette aux erreurs. Une alternative est de le produire automatiquement en se basant sur un Langage de Description d'Architecture Matérielle (*ADL*). Dans ce cas, plusieurs avantages se présentent : la mise en œuvre et les modifications de la plateforme sont plus rapides et la mise au point du simulateur, dans la mesure où il peut être généré automatiquement, est aussi plus rapide puisqu'une vérification de la cohérence de l'architecture peut être faite à partir de la description.

Un langage de description d'architecture matérielle permet de décrire tout ou partie des différents composants matériels qui forment un calculateur voire un ensemble de calculateurs communiquant au travers d'un réseau. Traditionnellement, les langages de description d'architecture ont pour objectif l'exploration de l'espace de conception. Ils permettent de décrire le jeu d'instructions du microprocesseur et l'architecture associée afin de produire des simulateurs du processeur et des outils logiciels comme un assembleur et un compilateur. Nous pouvons citer entre autres *nML* (*ADL* orienté jeu d'instructions), *MIMOLA* (*ADL* orienté structure), *LISA* et *EXPRESSION* (*ADL*s mixtes).

Cette thèse propose un nouveau langage de description d'architecture matérielle *HARMLESS* (*Hardware ARchitecture Modeling Language for Embedded Software Simulation*). C'est un *ADL* mixte ; il permet de décrire d'une manière concise les différentes parties d'un processeur : le jeu d'instructions et la structure interne (les composants matériels et le pipeline). L'originalité de *HARMLESS* est le découplage de la description du jeu d'instructions de la spécification de la micro-architecture (pipeline et concurrences d'accès aux différents composants matériels). L'une des conséquences est de permettre la génération des deux types de simulateurs indépendamment et simultanément : le simulateur de jeu d'instructions (*ISS*) permettant la vérification fonctionnelle du processeur et le simulateur précis au cycle près (*CAS*) fournissant des informations temporelles (en nombre de cycles) sur l'exécution de ce dernier. Une autre conséquence est une construction incrémentale de la description : 4 vues séparées permettent de décrire d'une part le jeu d'instructions (3 vues pour la syntaxe, le format binaire et la sémantique) et d'autre part la micro-architecture (une vue) du processeur. Ceci facilite la réutilisation du code sur une nouvelle architecture cible (les jeux d'instructions évoluent beaucoup moins vite que la structure interne d'un processeur). De nombreuses descriptions de processeur ont été réalisées pour prouver la

validité des concepts.

**Mots-clés :** systèmes temps réel, langage de description d'architecture matérielle ($ADL$), simulateur de jeu d'instructions ($ISS$), simulateur précis au cycle près ($CAS$), processeur.

# Abstract

With the development of mobile devices, embedded systems are increasingly used. With the rapid advances in technology, the architecture of programmable devices used to build such systems, processors, co-processors, memory, peripherals, becomes increasingly complex. In contrast, the lifetime of these systems and the time-to-market continues to decline. Therefore, we see the increasing need to automate the design of these programmable devices.

The hardware simulation is an important element in the design of embedded systems. A simulator can be hand written (eg. using a language like *C++*), but this task is complex, slow and error prone. An alternative is to generate it automatically based on a Hardware Architecture Description Language (*ADL*). In this case, several advantages arise : the implementation and modifications of the platform are faster and the development of the simulator, as far as it can be generated automatically, is also faster since the verification the consistency of architecture can be made from the description.

A hardware architecture description language allows to describe all or part of various hardware components that make up a computer or a set of computers communicating via a network. Traditionally, architecture description languages are intended to explore the design space. They describe the instruction set of the microprocessor and associated architecture to generate CPU simulators and software tools as an assembler and a compiler. We can mention among others *nML* (*instruction set ADL*, *MIMOLA* (*structural ADL*), *LISA* and *EXPRESSION* (*mixed ADL*s).

This thesis aims to propose a new hardware architecture description language *HARMLESS* (*Hardware ARchitecture Modeling Language for Embedded Software Simulation*). It is a *mixed ADL*; it allows to describe concisely the different parts of a processor : instruction set and internal structure (hardware components and pipeline). The originality of *HARMLESS* is the decoupling of the instruction set description from the micro-architecture description (pipeline and concurrency to access the different hardware devices). One consequence is to allow the generation of two types of simulators independently and simultaneously : the instruction set simulator (*ISS*) for the functional verification of processors and the cycle accurate simulator (*CAS*) providing temporal information (in cycles) on the performance of the latter. Another consequence is an incremental construction of description : 4 separate views to describe, one hand the instruction set (3 views for syntax, binary format and semantics) and other hand the processor micro-architecture (one view). This facilitates code reuse on a new target architecture (instruction set progresses much slower than the internal structure of a processor). Many processor descriptions were developed in order to prove the validity of the proposed concepts.

**Keywords :** real-time systems, hardware architecture description language (*ADL*), instruction set simulator (*ISS*), cycle accurate simulator (*CAS*), processor.

# Table des matières

# Table des figures

# Liste des tableaux

# Chapitre 1

# Introduction générale

De nos jours, les matériels informatiques sont présents dans de nombreux secteurs d'activités : les moyens de transport (automobile, aéronautique), la maison (ordinateur de bureau, machine à laver, etc), l'industrie, les universités, ... Ils sont devenus indispensables dans nos vies quotidiennes.

Dans ce travail, nous nous intéressons aux domaines d'applications nécessitant des systèmes embarqués (transport d'une manière générale) et plus particulièrement les systèmes qui ont des contraintes temps réel.

Cette thèse s'est déroulée au sein de l'équipe « Systèmes Temps Réel » de l'*IRCCyN* (*Institut de Recherche en Communications et Cybernétique de Nantes*).

Ces travaux ont été menés en partie en collaboration avec le projet *MasCotTE* (Maîtrise et Contrôle des Temps d'Exécution), financé dans le cadre du *PREDIT* 2005 (*Programme Transports Terrestre*), supporté par l'ANR (*Agence Nationale de la Recherche*.

## 1.1 Introduction aux systèmes temps réel

Chaque système informatique, mis en œuvre pour un domaine d'utilisation spécifique, possède des spécificités propres liées à ce domaine. Globalement, ces systèmes peuvent être classés en 3 catégories [28] :

- Systèmes *Transformationnels* : ils permettent d'élaborer un résultat à partir des données disponibles sans aucune contrainte sur le temps de réponse, comme les compilateurs ;
- Systèmes *interactifs* : ces systèmes interagissent avec l'environnement, et une réponse, dans un délai court, est souhaitable, comme les systèmes de réservation ;
- Systèmes *réactifs* : ces systèmes interagissent aussi avec leur environnement, mais le traitement suite à un événement survenu doit être accompli dans un

laps de temps prédéfini, comme c'est le cas pour les applications embarquées de commande/contrôle.

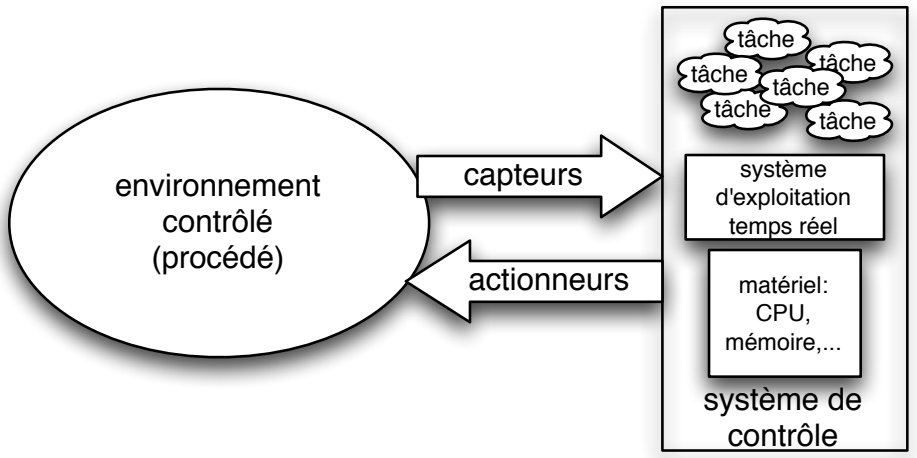



FIGURE 1.1 – Interaction entre un système de contrôle (temps réel) et l'environnement. Un tel système est composé d'un support d'exécution (matériel et logiciel) et d'applications.

Le terme réactif est souvent utilisé pour qualifier les systèmes temps réel, il permet de souligner deux concepts définissant ces systèmes [59] :

- Le premier se réfère aux interactions entre le procédé et le système informatique (voir figure 1.1) : via les capteurs le système de contrôle prélève des mesures, à intervalles réguliers, sur le procédé ; il reçoit également via les capteurs des informations de nature événementielle ; en retour le système de contrôle envoie au procédé des ordres (commandes) via les actionneurs ;
- Le second se réfère au temps. Un système temps réel doit réagir au bon moment, cela implique alors une observation et une action suffisamment rapides pour obliger le procédé à adopter le comportement souhaité ne lui laissant pas le temps d'évoluer de façon significative entre une variation d'état et l'action qui en résulte.

Traditionnellement, les systèmes temps réel peuvent être classés en trois catégories selon les différents niveaux de contraintes temporelles apposées sur ces systèmes :

- Les systèmes temps réel à contraintes souples (*soft real time*) : en cas de défaillance, la performance du système est dégradée sans engendrer des conséquences dramatiques. Par exemple, dans le multimédia, la violation

d'échéances entraîne uniquement des dégradations de Qualité de Service (*QoS : Quality of Service*) sans compromettre le bon fonctionnement du système et sans mettre en danger son intégrité ;
- Les systèmes temps réel à contraintes strictes (*hard real time*) : l'incapacité du système à respecter les échéances temporelles amène à une défaillance du système et peut entraîner des conséquences graves sur l'environnement contrôlé, comme les systèmes de contrôle aérien et le contrôle d'une centrale nucléaire ;
- Les systèmes temps réel à contraintes fermes (*firm real time*) : les contraintes sont sévères mais une faible probabilité de manquer les échéances temporelles peut être tolérée.

Dans tous les cas, un système temps réel est donc un système en interaction avec son environnement comme le montre la figure 1.1, dans lequel l'exactitude d'un calcul dépend non seulement du résultat logique (logical correctness) mais également de l'instant auquel le résultat est produit (timeliness) [57]. Le système doit ainsi être capable de réagir suffisamment rapidement pour que la réaction ait un sens. Par conséquent, une application temps réel implique généralement, dans son implémentation logicielle, des tâches auxquelles sont associées des contraintes d'échéances temporelles. Du fait qu'une défaillance du système peut avoir des conséquences importantes à l'échelle économique, humaine, environnementale... il y a la nécessité de la vérification et de la validation des systèmes temps réel, tant sur les aspects fonctionnels que sur les aspects temporels ainsi que la sûreté de fonctionnement. Cette vérification doit conclure à la prédictibilité [1] du système surtout dans le cas des systèmes temps réel stricts.

## 1.2 Validation des systèmes temps réel

En raison de leur criticité, la validation des systèmes temps réel est une tâche indispensable mais aussi complexe. Plusieurs techniques de validation existent. Ces techniques, souvent complémentaires, interviennent tout au long du cycle de développement de tels systèmes. Nous distinguons : la validation sur modèle d'une part et la validation sur cible d'autre part.

### 1.2.1 Validation sur modèle

La validation sur modèle englobe, à son tour, plusieurs approches complémentaires comme l'analyse formelle (par les automates ou les réseaux de

1. « Un système est dit prédictible [58] s'il est possible de prouver que les exigences fonctionnelles, temporelles et de sûreté de fonctionnement sont garanties vis à vis des hypothèses utilisées pour la conception du système, par exemple les hypothèses de défaillance ou de surcharge » [59].

Petri), l'analyse d'ordonnançabilité qui se fait sur un modèle analytique et la simulation.

#### 1.2.1.1 Analyse formelle

L'analyse formelle d'un système temps réel permet, à partir d'une modélisation à un haut niveau d'abstraction, d'obtenir des informations exhaustives. Elle permet la vérification des propriétés du système étudié, comme l'atteignabilité (qui énonce qu'une certaine situation peut être atteinte), la sûreté (qui énonce qu'une certaine situation ne peut jamais être atteinte sous certaines conditions) et la vivacité (qui énonce que quelque chose va se produire sous certaines conditions), à partir d'une représentation formelle comme, par exemple, les réseaux de Petri T-temporisé et les automates temporisés prenant en compte les aspects temporels d'un système.

L'intérêt principal de cette approche est qu'une propriété vérifiée sera vraie pour tous les scénarios possibles. Nous parlons alors de la vérification de modèles (ou *model-checking*) [10]. Elle consiste à construire, à partir d'une description formelle d'un système, l'espace des états atteignables. Puis, elle permet, en utilisant des algorithmes de parcours de l'espace des états, de vérifier le respect des propriétés attendues du système. C'est une technique effective de vérification automatique permettant de s'assurer que le système est exempt d'erreurs de conception.

L'analyse formelle permet donc de vérifier, pendant la phase de conception des systèmes temps réel, des propriétés concernant surtout la sécurité. Cependant, cette approche se heurte au problème de l'explosion combinatoire du nombre des états du système. En effet, les modèles utilisés ne doivent pas être trop complexes pour éviter les temps de calcul et les besoins en mémoire trop importants. De plus, se pose la question de la distance très importante entre le modèle haut niveau et son implémentation, et donc de la confiance que nous pouvons placer dans cette vérification, sur les aspects temporels.

#### 1.2.1.2 Analyse d'ordonnançabilité

Classiquement, une application temps réel est décomposée en un ensemble de tâches qui sont exécutées concurremment par un ou plusieurs processeurs. En effet, dans le cas d'un système monoprocesseur, ces tâches doivent se partager le processeur ; dans un système réparti composé de plusieurs processeurs interconnectés par un réseau, même si un certain degré de parallélisme existe, ces tâches doivent parfois coopérer et par suite se synchroniser temporellement. Par conséquent, selon l'architecture matérielle utilisée et selon les types d'applications (périodiques, sporadiques, cycliques,...), le comportement global du système temps réel peut devenir difficile à appréhender. De ce fait, il est nécessaire d'analyser le comportement temporel des tâches de l'application temps réel. Nous parlons alors de

l'analyse d'ordonnançabilité effectuée en utilisant différentes méthodes d'ordonnancement. Cette analyse est conduite pour une configuration de tâches et un algorithme d'ordonnancement. Selon le cas, le modèle utilisé pour les tâches est plus ou moins complexe incluant des grandeurs telles que : la période, le temps minimum d'exécution, l'offset au démarrage, l'échéance, des relations de précédence... Les méthodes utilisées vont de la simple application de formules analytiques pour des configurations simples à l'utilisation d'algorithmes de calcul de temps de réponse pour des configurations plus complexes. Cette discipline conduit à de très nombreux travaux [13].

Les méthodes d'ordonnancement permettent de définir l'ordre d'exécution des tâches sur le ou les processeurs, dans le but de prouver que les tâches respectent bien les contraintes temporelles et sont donc ordonnançables. Nous pouvons distinguer deux types de méthodes d'ordonnancement :

- les méthodes d'ordonnancement hors ligne : Dans ce cas, toutes les tâches et leurs paramètres temporels (date d'activation, date d'échéance,...) sont supposés connus avant le démarrage de l'application [44]. Il est alors possible de construire, avant l'exécution effective de l'application, une séquence valide vérifiant que toutes les contraintes temporelles sont satisfaites. Cette séquence sera chargée ensuite dans une table utilisée par le séquenceur tout au long de la vie de l'application. Les tâches seront donc exécutées dans l'ordre préétabli sans que cet ordre puisse être modifié. Ce type d'ordonnancement est surtout utilisé dans le cas des tâches périodiques.
- les méthodes d'ordonnancement en ligne : Dans ce cas, les décisions sur l'activation des différentes tâches sont prises par l'ordonnanceur dynamiquement, c'est-à-dire au moment de l'exécution de l'application temps réel. En effet, le principe de cette approche est basé sur le fait qu'un algorithme est implémenté directement au niveau de l'ordonnanceur qui, à chaque exécution, va intervenir et choisir la tâche à exécuter en utilisant un critère tel que la priorité d'une tâche. Pour ce type d'ordonnancement l'analyse d'ordonnançabilité doit bien entendu être effectuée hors ligne sur la configuration, et pour l'algorithme d'ordonnancement utilisé.

Quelle que soit la stratégie utilisée, la vérification et les garanties qu'elle est sensée apporter ne seront réalistes que si les paramètres de la configuration sont correctement déterminés, notamment le pire temps d'exécution (*WCET*, pour *Worst Case Execution Time*).

#### 1.2.1.3 Analyse par simulation

Une autre approche, complémentaire, utilisée dans le but d'analyser les performances, du dimensionnement ou de la mise au point des systèmes temps réel, est la simulation. Elle peut s'envisager sur des modèles plus ou moins fins du

système à analyser. Dans ce contexte, un ensemble de scénarios sera appliqué à un modèle dans un certain formalisme (Matlab/Simulink, C++, VHDL,...). L'exécution de chaque scénario permet d'obtenir des informations nécessaires à l'analyse du système en question. Un moteur de simulation produit des événements recevables par le modèle du système, qui à son tour va réagir à ces événements en changeant d'état et en émettant des événements de sortie, marqués par la date simulée. Deux approches de simulateurs existent :

- Simulation à temps continu (comme dans Matlab/Simulink) : dans cette approche, le moteur de simulation avance d'un pas fixe (période d'échantillonnage). L'état du système sera alors rafraîchi périodiquement, et ceci indépendamment de l'état du modèle ;
- Simulation à temps discret (comme dans SystemC) : dans ce cas, le moteur de simulation avance le temps de simulation, après que tous les événements de sorties sont émis, jusqu'à l'occurrence d'un nouvel événement. Cet événement sera injecté dans le modèle et ainsi de suite.

Nous nous intéressons ici uniquement au cas de la simulation à temps discret. Par opposition aux autres approches sur modèle d'analyse des systèmes temps réel qui ne permettent pas une modélisation très fine de tels système en raison de l'explosion combinatoire par exemple, l'analyse par simulation donne la possibilité de simuler les systèmes beaucoup plus finement puisque le temps de simulation est affecté dans une proportion acceptable par la complexité du modèle à simuler. Par contre, il faut choisir intelligemment les scénarios à simuler en raison de leur nombre souvent important, pour faire apparaître les cas dans lesquels les contraintes sur le système simulé sont les plus fortes, ou bien les générer automatiquement permettant ainsi d'avoir une répartition la plus large possible de l'espace des scénarios.

Suivant les objectifs attendus en terme d'observation, plusieurs niveaux de simulation sont possibles. Nous nous focalisons ici sur deux niveaux :

**Simulation au niveau système d'exploitation :** dans ce cas, la simulation est faite à un haut niveau d'abstraction offrant une vue globale du système [15] d'une part et permettant de détecter les erreurs de conception au plus tôt dans le cycle de développement d'autre part. En effet, dans cette approche, les appels système à l'exécutif temps réel (système d'exploitation temps réel) sont modélisés. Le ou les processeurs et les bus de communication sont modélisés sous forme de ressources partagées, dont l'occupation sera gérée, pendant la simulation, par l'ordonnanceur. Dans cet environnement, la simulation est rapide et permet d'obtenir des informations de haut niveau sur le comportement opérationnel du système : les prises de ressources, les synchronisations, la gestion des interruptions, ...

**Simulation du support matériel :** dans ce cas, la finesse de modélisation des

différents éléments composant le support matériel (le ou les processeurs et les réseaux) définit la précision de la simulation [6]. En effet, dans cette approche, le support matériel est modélisé, mais par contre l'exécutif temps réel ainsi que les tâches applicatives ne seront pas modélisés ; ils seront directement exécutés (leur code) sur le support matériel durant la simulation. De ce fait, la vitesse de simulation dépend de la finesse de modélisation, et les informations obtenues sont de bas niveau et avec une grande précision temporelle.

### 1.2.2 Validation sur cible

Ce type de validation est utilisé généralement en fin du cycle de développement du système temps réel. Il permet de tester le produit final, par expérimentation par exemple, afin de s'assurer qu'il :

- offre le niveau de qualité requis par le cahier des charges et satisfait bien les exigences fonctionnelles et techniques qui ont guidé sa conception et son développement ;
- est exempt des défauts et fonctionne de manière optimale.

Par conséquent, la procédure de test sur cible peut être plus ou moins fine et précise. Ainsi, l'effort de test sera plus ou moins important et coûteux selon le niveau de qualité requis.

*Dans ce travail, nous nous intéressons à la simulation du support matériel tout en restant à un niveau d'abstraction plus élevé que celui des simulateurs obtenus à partir des langages de description matériel comme VHDL, c'est-à-dire que nous ne modélisons pas, par exemple, les connexions entre les différents composants matériels. Notre but n'est en effet pas de synthétiser un processeur, mais de valider temporellement une application temps réel associée à son système d'exploitation.*

## 1.3 Langage de description d'architecture matérielle (*HADL*)

Construire un simulateur à la main est une tâche longue et difficile surtout lorsque l'architecture du processeur devient complexe. Cette tâche peut être atténuée en utilisant un langage de description d'architecture matérielle (*HADL*[2], pour *Hardware Architecture Description language*) qui permettra alors (c'est un objectif) de générer automatiquement le simulateur.

2. Souvent *ADL* est utilisé pour désigner également un langage de description d'architecture matérielle.

Un ADL est un moyen de description commun à l'architecte matériel et au concepteur d'outils pour une même architecture. Le premier facteur ayant favorisé le développement des langages ADLs est l'avènement des *ASIP*s (*Application-Specific Instruction set Processors*). Le second facteur est l'intérêt grandissant pour les *DSP*s (*Digital Signal Prosessors*) ainsi que pour les *SoC*s (*System-on-Chips*).

Des langages tels que *VHDL* ou *VERILOG* permettent déjà de décrire des architectures matérielles. Cependant, un ADL se situe à un niveau d'abstraction plus élevé et permet :

- la vérification sur l'architecture ;
- des modifications rapides sur l'architecture ou le jeu d'instructions pour évaluer les performances (pour l'exploration matérielle lors de la phase de conception : *Design Space Exploration*) ;
- l'obtention d'un modèle de processeur plus rapidement (le niveau de finesse est moins élevé, tous les bus ne sont pas forcément modélisés).

Une introduction plus détaillée sur les ADLs est donnée dans la chapitre 2.

## 1.4 Les architectures matérielles

Comme nous l'avons déjà précisé, ce travail de thèse se focalise sur la simulation du support matériel. Les architectures matérielles utilisées dans les systèmes temps réel peuvent être plus ou moins complexes allant d'une architecture simple contenant un seul processeur à une architecture plus complexe composée de plusieurs processeurs partageant ou non la mémoire. Nous distinguons alors trois grandes catégories d'architecture matérielle pour de tels systèmes :

**L'architecture monoprocesseur :** cette architecture comporte un processeur unique s'occupant de l'exécution de toutes les tâches applicatives concurrentes. Le temps processeur est alors partagé entre toutes ces tâches. De ce fait, il n'y a donc pas du vrai parallélisme d'exécution mais un pseudo-parallélisme ou entrelacement des exécutions ;

**L'architecture multiprocesseur :** cette architecture comporte plusieurs processeurs (identiques ou non) partageant une unique mémoire centrale. Dans ce cas, l'exécution des tâches applicatives sera répartie sur les différents processeurs, d'où un vrai parallélisme d'exécution. Ces tâches coopèrent entre elles en partageant des informations via la mémoire centrale ;

**L'architecture répartie :** comme dans le cas d'une architecture multiprocesseur, elle comporte aussi plusieurs processeurs mais sans mémoire commune. Ces processeurs, chacun possédant sa propre mémoire et éventuellement un coprocesseur flottant ou vectoriel, sont reliés entre eux par des réseaux leur permettant de coopérer entre eux par envoi de message.

Pour la suite nous considérons le cas des architectures monoprocesseur. Comme nous l'avons déjà évoqué dans la section 1.1, une propriété importante qu'un système temps réel, surtout strict, doit vérifier est la prédictibilité. En effet, elle peut être affectée par plusieurs facteurs [9] : les caractéristiques architecturales du matériel, les mécanismes et les politiques adoptées dans le noyau temps réel et même par le langage de programmation utilisé pour implémenter une application. Nous nous focalisons, ici, sur les caractéristiques architecturales du matériel.

D'une manière générale, l'architecture matérielle d'un système temps réel est composée en plus d'un calculateur, des capteurs et des actionneurs. Un calculateur est composé d'une unité centrale de traitement (le processeur ou *CPU*, pour *Central Processing Unit*), des mémoires, une horloge pour cadencer le CPU et des unités de communications permettant de communiquer avec le monde extérieur (les périphériques externes, les capteurs, les actionneurs, ...). Un circuit intégré incluant un CPU, des mémoires, des unités périphériques et des interfaces d'entrées-sorties est un microcontrôleur. Les domaines d'applications typiques des microcontrôleurs sont les systèmes embarqués, et ceci en raison de leur faible consommation et de l'intégration de toutes les fonctions matérielles nécessaires à la réalisation d'un système de contrôle/commande.

L'élément matériel principal affectant la prédictibilité de l'ordonnancement des différentes tâches applicatives des systèmes temps réel est le processeur. En effet, les caractéristiques internes du processeur comme l'existence d'un pipeline et la mémoire cache sont les premières causes du non déterminisme (bien que le programme soit lui-même déterministe). Un autre élément matériel affectant la prédictibilité de tels systèmes est la mémoire. En effet, la hiérarchisation de la mémoire et l'unité de gestion mémoire (ou *MMU*, pour *Memory Management Unit*) dont l'utilisation la plus courante est la protection de plages mémoire sont aussi les causes du non déterminisme.

*Dans la section suivante, nous nous focalisons sur la présentation des pipelines qui est au cœur de nos travaux de thèse. Il est difficile d'étudier le comportement temporel d'un processeur contenant un pipeline sans modéliser le fonctionnement de ce dernier qui, comme nous l'avons évoqué plus haut, est une source non négligeable d'indéterminisme. En effet, le temps d'exécution d'une instruction dépend des instructions déjà présentes dans le pipeline.*

## 1.5 Le pipeline

### 1.5.1 Le concept de base du pipeline

Dans un processeur sans pipeline, les instructions d'un programme sont exécutées sans recouvrement les unes à la suite des autres. Une instruction ne peut commencer son exécution avant que l'instruction précédente ne soit entièrement exécutée. Supposons, par exemple, que l'exécution d'une instruction peut être découpée en 4 étapes élémentaires et qu'un cycle d'horloge est nécessaire pour chacune de ces étapes :

**Fetch (*F*) :** cette étape recupère le code de l'instruction de la mémoire ;

**Decode (*D*) :** cette étape permet le décodage de l'instruction ;

**Execution (*E*) :** cette étape se charge de l'exécution de l'instruction ;

**Write Back (*W*) :** cette étape s'occupe de stockage des résultats dans les registres.

Sur un processeur non pipeliné, l'exécution des instructions se fait selon le diagramme temporel de la figure 1.2. En effet, quand la première instruction commence son exécution et entre dans l'étape *Fetch*, la deuxième instruction ne peut pas s'exécuter avant que la première n'achève l'étape *Write Back*, et ainsi de suite. De cette manière, l'exécution de 3 instructions, par exemple, nécessitent 12 cycles d'horloge, ceci représente le débit d'instructions du processeur. De plus, dans ce cas, le nombre moyen de cycles par instructions (le *CPI* pour *Cycle Per Instruction*) est égal à 4 (1 cycle d'horloge par étape).

Dans le but d'optimiser le fonctionnement du processeur et ainsi augmenter son débit d'instructions et diminuer le *CPI*, une technique d'implémentation, dans laquelle des instructions peuvent être exécutées en parallèle (c'est-à-dire en parallélisant les étapes élémentaires citées plus haut), est utilisée. Cette technique est connue sous le nom de technique de recouvrement d'instructions, et est obtenue en utilisant l'élément architectural appelé *pipeline*. En effet, dans un pipeline, chaque étape élémentaire constitue un étage. L'indépendance de ces étages permet au processeur d'exécuter simultanément plusieurs instructions (chacune se trouvant dans un étage du pipeline) et ainsi d'augmenter le débit d'instructions (c'est-à-dire que le programme à exécuter va plus vite mais une instruction individuelle prend bien sûr autant de cycles d'horloges que dans le cas d'un processeur sans pipeline). En d'autres termes, cette technologie permet de réduire le *CPI* qui peut être idéalement égal à 1.

Prenons, par exemple, un pipeline à 4 étages (Chaque étage correspond à une des étapes élémentaires comme le montre la figure 1.3). En supposons toujours que chaque étage prend un cycle d'horloge, nous pouvons constater que dans le cas d'un processeur pipeliné, l'exécution de 3 instructions nécessite 6 cycles d'horloges

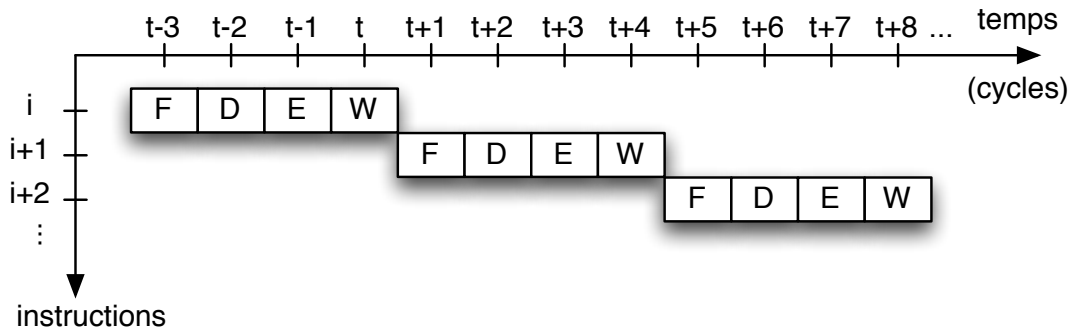



FIGURE 1.2 – Exécution des instructions dans un processeur sans pipeline. 12 cycles d'horloge sont nécessaires pour exécuter 3 instructions.

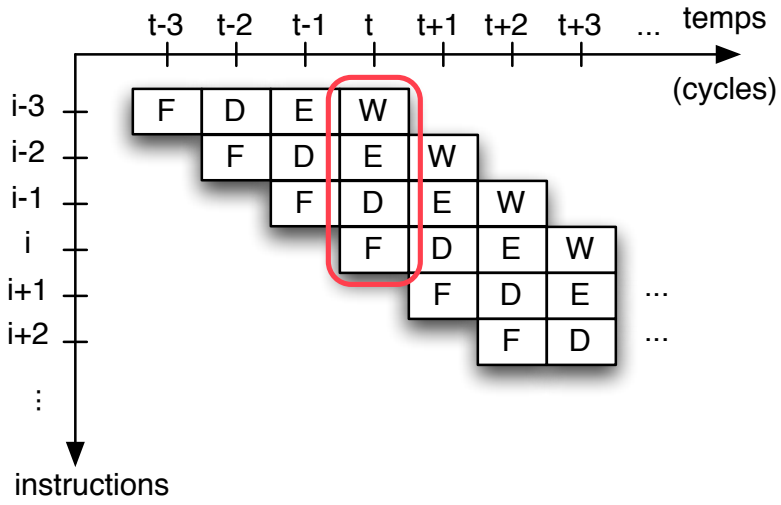



FIGURE 1.3 – Exécution des instructions dans un processeur contenant un pipeline. 7 cycles d'horloge sont nécessaires pour exécuter 3 instructions indépendantes.

contre 12 cycles dans le cas d'un processeur sans pipeline. En effet, quand une instruction quitte l'étage *Fetch* et entre dans l'étage *Decode*, l'instruction suivante peut commencer son exécution en entrant dans le premier étage sans attendre que la première sorte complètement du pipeline et ainsi de suite. À partir de l'instant $t$, nous remarquons que tous les étages de pipeline sont en exécution (occupés par des instructions), ce qui permet d'obtenir un *CPI* égal à 1.

La figure 1.3 nous montre le principe de fonctionnement idéal d'un pipeline [29]. Cependant, dans la réalité, certaines situations peuvent venir perturber le

bon fonctionnement du pipeline réduisant ainsi sa performance, ce sont *les aléas*.

### 1.5.2 Différentes catégories d'aléas

Les aléas peuvent influer sur l'accélération idéale qu'offre le pipeline, puisqu'ils retardent l'exécution d'une ou plusieurs instructions. Ils sont classés en trois catégories :

- les *aléas structurels* résultent d'un manque de ressources matérielles. Le processeur n'arrive pas à gérer correctement le recouvrement des instructions en cours d'exécution. Par exemple, deux instructions (dans deux étages différents du pipeline) essaient d'accéder à la mémoire simultanément en écriture et en lecture, ce qui occasionne un conflit qu'il faut résoudre (voir annexe B.1)
- les *aléas de données* interviennent quand il y a des dépendances de données entre les instructions. Prenons, par exemple, le pipeline à 4 étages de la figure 1.3. Supposons que l'instruction *ADD R1, R2, R3* est dans l'étage `F` et l'instruction *SUB R2, R5, R6* est dans l'étage `D`. L'instruction *ADD* a besoin de *R2* et *R3* pour entrer dans l'étage `D`, mais le registre *R2* sera mis à jour par l'instruction précédente *SUB* dans l'étage `W`. Donc, l'instruction *ADD* sera bloquée dans l'étage `F` à cause de l'aléa de données jusqu'à ce que l'instruction *SUB*, écrit le résultat dans l'étage `W`; (voir annexe B.2)
- les *aléas de contrôle* surviennent à cause des instructions de sauts (branchements avec ou sans condition, boucles, appels de fonctions, ...). En d'autres termes, toutes les instructions qui modifient le compteur programme *PC* (*Program Counter*). (voir annexe B.3)

En cas d'aléas, le pipeline peut être suspendu, ceci retarde l'avancement de l'exécution des instructions et nécessite d'insérer des suspensions (bulles) pour que ce pipeline puisse avancer correctement. Il y a plusieurs méthodes pour gérer de la manière la plus efficace ces aléas comme par exemple la prédiction de branchement (afin de réduire les pénalités des aléas de contrôle, voir section 6.4.1), l'exécution spéculative (exécution anticipée d'une instruction sans être certain que celle-ci ait réellement besoin d'être exécutée pour réduire les pénalités causées par les instructions de branchement conditionnel), l'ordonnancement dynamique (exécuter les instructions d'un programme dans le désordre afin de réduire les suspensions causées par les aléas de données), le renommage de registres (permet de diminuer les aléas de données) ...

## 1.6 Motivation et objectifs de la thèse

Avec les progrès rapides de la technologie, l'architecture des composants programmables (comme les processeurs, les co-processeurs et les périphériques), utilisés dans les systèmes embarqués temps réel, devient de plus en plus complexe. D'où la nécessité d'une automatisation de la conception de ces composants en utilisant des modèles de niveau d'abstraction plus ou moins élevé.

Les simulateurs écrits à la main ont montré un point faible important. En effet, un changement de cible d'exécution nécessite une réécriture complète d'un tel simulateur, ce qui représente un travail important, tant au niveau de la conception que de la mise au point.

Pour pallier ce problème, nos travaux sur ce sujet se concentrent sur la génération automatique du simulateur à partir d'une description de son comportement, via un Langage de Description d'Architecture matérielle nommé *HARMLESS* (*Hardware ARchitecture Modeling Language for Embedded Software Simulation*). L'intérêt de la génération automatique du simulateur est de réduire le temps de développement nécessaire à l'obtention d'un simulateur. De plus, le nombre d'erreurs dans le simulateur devrait être largement inférieur à un développement *ad hoc*.

Traditionnellement, l'objectif des langages de description d'architecture matérielle est l'exploration de l'espace de conception. Ces langages permettent, d'une part, une description concise du jeu d'instructions du microprocesseur et d'autre part, la description de l'architecture associée afin de produire des simulateurs du processeur. Plusieurs ADLs existent, comme *nML* (*ADL* orienté jeu d'instructions), *MIMOLA* (*ADL* orienté structure), *LISA* et *EXPRESSION* (*ADL*s mixtes).

Or, aucun des langages que nous avons étudiés (voir chapitre 2) et prenant en compte l'architecture interne des processeurs (*EXPRESSION* ou *LISA* notamment) ne convient. *LISA*, par exemple, ne permet pas, de générer indépendamment et simultanément les deux types de simulateurs : le simulateur de jeu d'instructions (*ISS*, pour *Instruction Set Simulator*) utilisé pour la vérification fonctionnelle d'un programme et le simulateur précis au cycle près (*CAS*, pour *Cycle Accurate Simulator*) fournissant des informations temporelles, en nombre de cycles, sur l'exécution de ce dernier. En effet, la modélisation du fonctionnement du pipeline n'est pas indépendante de la description du jeu d'instructions (voir section 2.5.1). Les instructions doivent être assignées explicitement à chaque étage de pipeline. Par conséquent, en cas de changement dans la description du pipeline, la description du jeu d'instructions doit être réécrite. Les deux ADLs n'offrent pas la possibilité de décrire les instructions de taille variables.

Enfin, Notre approche n'a pas pour but de permettre la synthétisation de processeurs comme *EXPRESSION* (voir section 2.5.2), mais uniquement d'en décrire

les aspects temporels. Dans ce contexte, une modélisation du comportement du ou des pipeline(s) par des automates d'états finis est faite. Cette approche déporte une part importante du calcul au moment de la synthèse du simulateur. L'utilisation d'automates permet ainsi d'obtenir des performances de simulation très élevées.

## 1.7 Organisation du mémoire

Cette mémoire se compose au total de **huit chapitres**, suivi de **trois annexes**.

Après avoir défini brièvement les systèmes temps réel, exposé les grandes lignes des différentes méthodes d'analyse de ces systèmes ainsi que l'architecture matérielle de tels systèmes dans **ce chapitre** « Introduction générale », **le deuxième chapitre** est consacré à l'état de l'art. Tout au début, une classification des ADLs selon leur contenu (orientés jeu d'instructions, orientés structure interne ou mixtes) d'une part et selon leur objectif (compilation, simulation, ...) d'autre part sera donnée. Ensuite, nous expliquerons les différents ADLs existants. Enfin, nous allons positionner nos travaux par rapport aux ADLs présentés.

Les troisième et quatrième chapitres constituent **la première partie** de notre travail qui se focalise sur le simulateur de jeu d'instructions.

**Le troisième chapitre** concerne la description d'une architecture fonctionnelle dans HARMLESS. Pour engendrer un simulateur fonctionnel, il faut décrire, d'une part, le jeu d'instructions du processeur et d'autre part, les composants matériels (description fonctionnelle). Dans ce cas, il n'est pas nécessaire de décrire la micro-architecture du processeur (pipeline et concurrences d'accès aux différents composants notamment) qui sera obligatoire dans la génération du simulateur précis au cycle près. La description du jeu d'instructions se décompose en trois vues :

- une vue dédiée à la description du format binaire ;
- une vue pour décrire la syntaxe textuelle des instructions ;
- une vue permettant de décrire le comportement du jeu d'instructions.

La décomposition en trois vues permet une description progressive et souple du jeu d'instructions. Les composants matériels sont décrits séparément de la description du comportement des instructions dans le but de pouvoir générer en plus un simulateur précis au cycle près. Dans ce dernier cas, des contraintes temporelles seront associées à l'utilisation des ressources matérielles par les instructions.

Dans **le quatrième chapitre**, nous exposerons, dans un premier temps, les différentes phases nécessaires à la génération du simulateur de jeu d'instructions en passant par la modélisation d'instructions, la génération du décodeur et l'importance d'un cache logiciel du point de vue de la vitesse de simulation.

Dans un second temps, les résultats portant sur la génération d'un simulateur de jeu d'instructions pour plusieurs processeurs (le *PowerPC*, le *HCS12* et son co-processeur le *XGate* et l'*AVR*) seront analysés.

Les cinquième, sixième et septième chapitres constituent **la deuxième partie** de notre travail qui se focalise sur le simulateur précis au cycle près.

**Le cinquième chapitre** se focalise sur la modélisation des aspects architecturaux dans le but de générer automatiquement un simulateur précis au cycle près. Après avoir expliqué brièvement quelques travaux portant sur la modélisation du fonctionnement du pipeline en utilisant soit les automates soit les réseaux de Petri, nous allons introduire notre approche qui consiste à modéliser le fonctionnement d'un pipeline par un automate à états finis permettant, contrairement aux autres travaux, de traiter tous les types d'aléas (les aléas structurels, les aléas de données et les aléas de contrôle) soit statiquement (durant la compilation) via les ressources internes soit dynamiquement (durant la simulation) via les ressources externes (les ressources qui peuvent être prises, non seulement par les instructions, mais aussi par des périphériques, par exemple). De plus, les 2 outils *p2a* et *a2cpp*, permettant de générer respectivement l'automate à états finis et le code C++ correspondant, seront présentés. Finalement, deux exemples permettront de bien comprendre notre approche.

Dans **le sixième chapitre**, nous présenterons comment les éléments de la description fonctionnelle sont mis en correspondance avec le modèle interne (le pipeline) du processeur. Tout d'abord, nous exposerons comment HARMLESS permet la description de la vue micro-architecture d'un processeur : le pipeline ainsi que les concurrences d'accès aux composants matériels permettant de déduire les caractéristiques temporelles. Ensuite, nous présenterons le mappage de la vue sémantique du jeu d'instructions sur la vue micro-architecture en utilisant les accès aux composants matériels, ce qui est nécessaire afin de pouvoir simuler temporellement le processeur. Puis, la gestion des branchements dans notre langage sera expliquée tout en parlant brièvement des méthodes de prédiction de branchement existantes. Enfin, nous nous intéresserons, dans HARMLESS, à la méthode de découpage d'un pipeline en plusieurs pipelines de plus faible profondeur, permettant ainsi de générer des automates plus petits, et à l'avantage qu'elle apporte : résolution du problème de l'explosion combinatoire de l'automate.

**Le septième chapitre** porte sur la partie du compilateur s'occupant de la génération du simulateur précis au cycle près, à partir de la description fonctionnelle et temporelle du processeur. Ainsi, nous présenterons la génération des deux types de classes d'instructions. Le premier permet de regrouper dans une même classe les instructions qui utilisent les mêmes ressources internes et externes permettant de réduire l'espace d'états de l'automate (le modèle du pipeline).

Le second permet de classer ensemble les instructions qui font les mêmes accès en lecture et en écriture au banc de registres. De plus, nous expliquerons la technique d'exécution des instructions à travers les étages du pipeline entier ou découpé. Finalement, nous analyserons les résultats portant sur la génération d'un simulateur précis au cycle près en changeant la profondeur de pipeline tout en se basant sur le processeur *PowerPC 5516*.

**Le huitième chapitre** est une conclusion générale qui rappelle les grandes lignes des contributions de cette thèse, suivi d'un ensemble de perspectives, à plus ou moins long terme, permettant de donner une suite à ces travaux.

Ce mémoire contient trois annexes. **L'annexe A** présente un exemple détaillé sur la description de quelques instructions, permettant ainsi une vue d'ensemble des trois vues (le format binaire, la syntaxe et le comportement du jeu d'instructions).

**L'annexe B** détaille les différents types d'aléas (aléas structurels, aléas de données et aléas de contrôle) qui peuvent causer la suspension d'un pipeline.

**L'annexe C** présente le fichier spécifiant les caractéristiques d'un pipeline (comme les étages, les classes d'instructions, ...) décrit dans HARMLESS.

# Chapitre 2

# État de l'art

## 2.1 Introduction

Les langages de description d'architecture matérielle (*ADL*s) ont pour but de décrire de manière formelle ou semi-formelle les architectures logicielles ou matérielles.

Beaucoup de travaux ont été réalisés sur les *ADL*s pour pouvoir décrire un très large spectre de processeurs existants sur le marché. *VHDL* et *VERILOG* sont des langages permettant déjà de décrire des architectures matérielles. Ils sont largement répandus et utilisés pour la modélisation et la simulation des processeurs principalement dans le but de concevoir une nouvelle architecture matérielle. Toutefois, le niveau d'abstraction d'un ADL se situe à un niveau plus élevé et permet de générer d'une manière automatique et rapide un simulateur du processeur ou encore un assembleur, parfois un éditeur de lien et un déboggeur (*ADL LISA*), mais ce n'est pas l'objectif poursuivi par notre *ADL* HARMLESS.

Dans ce chapitre nous allons exposer, dans un premier temps, la classification des *ADL*s du point de vue *contenu* ainsi que de point de vue *objectif*. Ensuite, plusieurs *ADL*s existants seront abordés. Enfin, le positionnement de nos travaux vis à vis des autres ADLs sera donné.

## 2.2 Classification des ADLs

Dans cette section, nous allons classer les ADLs dans le contexte de la conception des processeurs embarqués [40]. Les ADLs peuvent être classés selon deux aspects : *contenu* et *objectif*, comme le montre la figure 2.1. La classification orientée *contenu* est basée sur la nature des informations qu'un ADL peut capturer, en d'autres termes le niveau de modélisation autorisé : comportemental, structurel ou les deux ensemble. La classification orientée *objectif* est quant à elle basée sur

le but visé par un ADL. Cependant, il n'est pas toujours possible d'établir une correspondance directe entre les deux types de classifications.

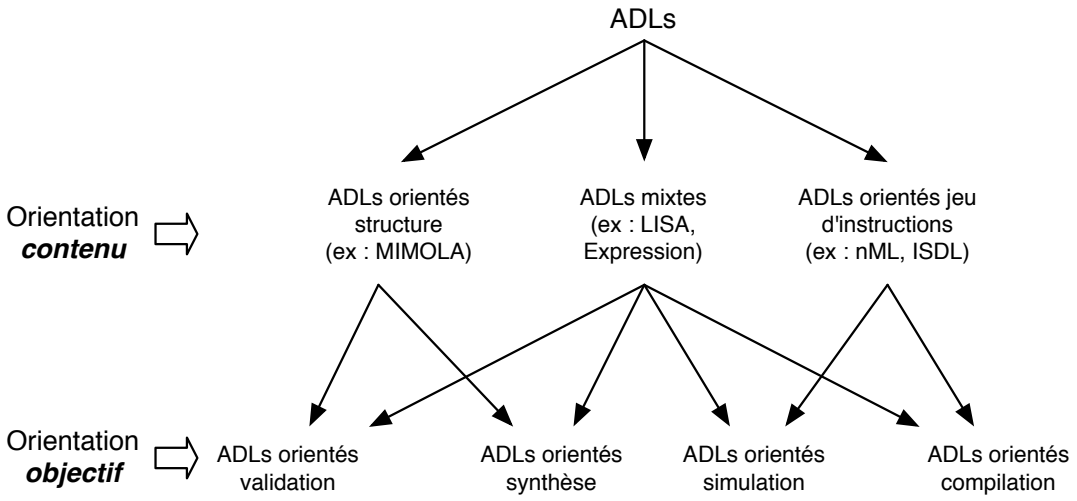



FIGURE 2.1 – Classification des ADLs selon les deux aspects : *contenu* et *objectif*.

### 2.2.1 Classification basée sur le contenu

Les ADLs peuvent être classés en trois catégories selon qu'ils capturent la structure interne du processeur (les composants matériels et leur connectivité), le comportement du jeu d'instructions, ou bien les deux ensembles.

#### 2.2.1.1 ADLs orientés jeu d'instructions

Cette catégorie se focalise sur les aspects jeu d'instructions du processeur en négligeant la structure matérielle détaillée, comme *nML* [17] et *ISDL* [24] par exemple. L'objectif annoncé de ce type d'ADL est de fournir, pour une architecture donnée, un compilateur ou/et un simulateur de type *ISS*[1]. Comme ce type d'ADL donne uniquement des informations sur le jeu d'instructions, il n'est pas conçu pour générer un simulateur précis au cycle près.

1. Simulateur fonctionnel, non précis temporellement

#### 2.2.1.2 ADLs orientés structure interne

Cette catégorie va plutôt décrire la structure interne de l'architecture à l'exemple de *MIMOLA* [4] qui permet de décrire les connexions entre les éléments composant l'architecture. Dans ce cas l'objectif est la production de code HDL [2]. Grâce aux informations contenues dans une description *MIMOLA*, il est possible de produire un simulateur mais ce n'est pas l'objectif principal. Grâce à ce type d'ADL, c'est la partie exploration matérielle (synthèse et validation) qui est privilégiée. Il est possible de considérer ce type d'ADL comme complémentaire du premier. En effet, pour produire un compilateur optimisé, il faut des informations sur la structure interne.

#### 2.2.1.3 ADLs mixtes

Cette catégorie captent les deux détails de l'architecture (la structure interne et le jeu d'instructions). *LISA* [62, 46] et *EXPRESSION* [26], entre autres, constituent des exemples de ce type d'ADL, qui tentent de couvrir les deux domaines. Ils fournissent une chaîne de développement qui permet l'exploration matérielle, la génération de code HDL mais aussi la génération d'outils.

### 2.2.2 Classification basée sur l'objectif

Les ADLs peuvent être classés en 4 catégories selon l'objectif visé :

#### 2.2.2.1 ADLs orientés synthèse

Les ADLs orientés structure interne (comme *MIMOLA*) ainsi que les ADLs mixtes (comme *LISA* et *EXPRESSION*) sont appropriés pour la production de matériel. Par exemple dans [55], Schliebusch et *al.* présentent un outil de synthèse qui préserve la flexibilité totale de l'ADL *LISA*, tout en étant capable de générer l'architecture complète à un niveau d'abstraction suffisamment bas (en anglais, *RT-level* pour *Register Transfer level*) utilisant *SystemC* [3].

#### 2.2.2.2 ADLs orientés validation

Les ADLs ont été utilisés dans l'industrie ainsi que dans les milieux académiques afin de permettre la génération de tests pour valider fonctionnellement les proces-

2. *Hardware Description Language* : c'est un langage de description des circuits électroniques. Il permet d'une part une description comportementale du circuit (les fonctions effectuées par le circuit) et d'autre part, une description structurelle (les composants utilisés par le circuit et leurs interconnexions) comme *VHDL* ou *Verilog*.

3. `http://www.systemc.org/`

seurs embarqués. Les ADLs structurels (comme *MIMOLA*) sont traditionnellement utilisés pour la génération de tests. Les ADLs mixtes, eux aussi, conviennent bien pour la génération de tests de validation, par exemple, la génération de tests automatisés en utilisant le modèle du processeur décrit par *LISA* [37].

#### 2.2.2.3 ADLs orientés compilation

Le but de ce type d'ADLs est de permettre la génération automatique des compilateurs « reciblables ». Un tel compilateur est conçu pour être relativement facile à modifier pour générer du code pour différentes architectures de processeurs. Les ADLs orientés jeu d'instructions ainsi que les ADLs mixtes sont adaptés pour la génération de compilateur.

#### 2.2.2.4 ADLs orientés simulation

La simulation peut être réalisée à différents niveaux d'abstraction. Au plus haut niveau d'abstraction, un simulateur de jeu d'instructions (simulateur fonctionnel ou *ISS*) peut être obtenu en modélisant uniquement le jeu d'instructions d'un processeur. Les ADLs orientés jeu d'instructions sont capables de générer ce type de simulateur. Par contre, la génération des simulateurs précis au cycle près (*CAS*) nécessite la description de l'aspect temporel du processeur (pipeline, contraintes sur l'utilisation des composants matérielles,...) qui se situe à un niveau d'abstraction plus bas. Les ADLs mixtes ainsi que les ADLs structurels permettent alors de générer ce type de simulateur. Basé sur le modèle de simulation, les simulateurs peuvent être classés en 3 types :

- Simulateur interprété (par exemple, *GENSIM/XSIM* [25] utilisant *ISDL* et *SIMPRESS* [32] utilisant *EXPRESSION*) : cette catégorie de simulateurs est basée sur un modèle interprété du jeu d'instructions d'un processeur. Il s'ensuit donc 3 étapes pour chaque instruction : extraction de la mémoire, décodage et exécution. Les simulateurs interprétés stockent l'état du processeur dans une mémoire hôte. De ce fait, ce type de simulateur est souple, facile à implémenter et à modifier. Par contre, il est plus lent par rapport aux autres approches car le processus de décodage des instructions consomment un temps non négligeable dans ce type de simulateur ;
- Simulateur compilé (par exemple, les travaux de Pees et *al.* [45] basé sur *LISA*) : cette approche réduit le temps d'exécution en transformant chaque instruction cible en une série d'instructions, sur la machine hôte, qui manipule l'état de la machine simulée. En d'autres termes, elle effectue le décodage des instructions pendant la compilation du programme d'application pour augmenter ainsi la vitesse de simulation. Cependant tous les simulateurs compilés supposent que le code du programme à exécuter est intégralement

connu avant que la simulation commence ;
- Simulateur mixte (par exemple, JIT-CCS [42]) : ce type de simulateur combine la souplesse des simulateurs interprétés avec la rapidité de simulation qu'offrent les simulateurs compilés.

Par la suite, nous allons présenter quelques ADLs, en sélectionnant les plus connus dans le domaine de la génération automatique des simulateurs de processeurs, notamment des désassembleurs et des compilateurs [39].

## 2.3 Les ADLs orientés jeu d'instructions

Comme nous l'avons déjà défini dans la section 2.2.1.1, les ADLs orientés jeu d'instructions permettent de décrire explicitement le comportement des instructions sans trop détailler la structure des composants matériels. Typiquement, il y a une correspondance directe entre la description faite en utilisant un tel type d'ADL et le manuel de référence du jeu d'instructions d'un processeur. Dans cette section, nous allons décrire brièvement 2 ADLs orientés jeu d'instructions : nML [18, 17], et ISDL [24].

### 2.3.1 nML

Le langage nML [18, 17] a été proposé à l'Université technique de Berlin, Allemagne. Les concepteurs de nML ont utilisé la technique de la hiérarchisation pour décrire les jeux d'instructions, permettant ainsi de mutualiser les parties communes entre les différentes instructions en les groupant ensemble.

Pour permettre la représentation hiérarchique du jeu d'instructions, ce dernier est décrit en tant qu'une grammaire attribuée [43]. La définition des instructions peut être vu comme un arbre de *et/ou*, où les règles *ou* permettent de lister les alternatives et les règles *et* permettent de décrire la composition d'une instruction (par exemple, ses opérandes). Chaque dérivation possible de l'arbre correspond à une instruction. Un nœud de l'arbre est appelé *op* (pour *operation*) et il peut représenter soit une instruction, soit une sous-instruction ou bien un groupe d'instructions (voir exemple suivant). À ce nœud sont attachés des attributs prédéfinis comme *image*, *syntax* et *action* pour permettre respectivement la description du format binaire (nécessaire pour le décodage), la syntaxe textuelle (nécessaire pour le désassemblage) et le comportement (nécessaire pour simuler l'exécution) des instructions.

Une description nML est formée des deux parties principales suivantes :
- description des éléments de mémorisation (mémoire, registres), et des modes d'adressages, comme le montre l'exemple suivant :

```
mem prog_mem[1024,16];

reg R[16,32];
reg PC alias R[15];

mode SRC = REG | IMM ...

mode REG (n:card ( 3) )  = R[n]
  syntax =  format ("R%d",n)
  image =  format ("%3b",n)
...
```

Cet exemple décrit une mémoire `prog_mem` de 1k avec des éléments de 16 bits, 16 registres de 32 bits chacun, le compteur programme `PC` associé au registre 15 et le mode d'adressage pour `SRC` qui peut être soit un registre (`REG`) ou bien un immédiat (`IMM`). La syntaxe textuelle ainsi que le format binaire pour le mode d'adressage `REG` sont également décrits.

– description du jeu d'instructions comme le montre l'exemple suivant décrivant une partie des instructions utilisant l'unité arithmétique et logique `alu_inst` :

```
op instruction = alu_inst | branch | ...

op num_action = add | sub | ...

op alu_inst (a:num_action, src:SRC, dst:DST)
  action {
   temp_src = src;  //temp_src et temp_dst sont des variables
                    //globales
   temp_dst = dst;
   a.action;
   dst = temp_dst;
  }
   syntax =  format ("%s %s, %s", a.syntax, temp_dst.syntax,
                            temp_src.syntax)
   image =  format ("%s%s%s", a.image, temp_dst.image,
                            temp_src.image)

op add ()
  syntax = "add"
```

```
    image = "00000"
    action = {tmp_dst = tmp_dst + tmp_src}

  op sub ()
    syntax = "sub"
    image = "00001"
    action = {tmp_dst = tmp_dst - tmp_src}
   ...
```

La technique de hiérarchisation adoptée par nML permet une description concise du jeu d'instructions d'un processeur dans le but de générer un simulateur fonctionnel. C'est un ADL simple et facile à apprendre. Par contre, il est difficile de modéliser explicitement les instructions de tailles variables, et il n'est pas possible d'effectuer des opérations sur les champs de bits (rotation, par exemple) pour décoder et désassembler correctement les instructions le nécessitant.

Plusieurs extensions de nML existent, nous pouvons citer entre autres les langages Sim-nML [48] et GLISS-nML [49]. Il a été utilisé aussi par le simulateur de jeu d'instructions Sigh/Sim [36] et le générateur de code CBC [16], et par l'environnement CHESS/CHECKERS [1] permettant la simulation d'un jeu d'instructions et la compilation automatique et efficace du logiciel. Dans ce qui suit, nous allons présenter brièvement GLISS-nML.

**GLISS-nML [49]** Ce langage est une extension de nML permettant la description d'une plus grande variété de jeux d'instructions en introduisant des concepts améliorant la souplesse de description et diminuant la taille du code source (à titre d'exemple, le taille du code source obtenu pour le simulateur de jeu d'instructions *ARM* 32 bits est passé de 154 Mo (utilisant nML) à 3,4 Mo (utilisant GLISS-nML) [49]). Il a été développé à l'IRIT, Toulouse.

GLISS-nML donne la possibilité de déclarer des variables temporaires locales, sous forme de pseudo-paramètres (variable `tmp` dans l'exemple suivant, pris de [50]), nécessaires pour la partie `action` de la description d'une instruction. Dans nML, ces variables doivent être déclarées en tant que variables globales. Il a permis aussi d'introduire, dans la description, des fonctions écrites en langage de programmation *C*, comme le montre l'exemple suivant :

```
//DESCRIPTION GLISS-nML
op ABC(tmp:card(32), i :card(4), ...)
  image =  format ("%0b%4b ...", tmp.image, i.image, ...)
  action = {
    tmp = foo(i);
  }
```

```
 //FONCTION EXTERNE
int foo(int i) {...}
```

D'autre part, Ratsiambahotra et *al.* ont introduit dans nML un nouvel attribut nommé `predecode` permettant le décodage dynamique des paramètres pour désassembler correctement les instructions. Ce nouvel attribut donne la possibilité de déterminer la syntaxe d'un paramètre en fonction de sa valeur ou de celle d'autres paramètres, ou encore de calculer la valeur réelle d'un paramètre encodé [50]. L'exemple suivant montre une utilisation de ce nouvel attribut :

```
mode immediat (imm :card(32), rot:card(4), val:card(8))
  predecode {
    imm =  coerce(32, val>>>2*rot) ;
  }
  syntax =  format (\#%d",imm)
  image =  format (\%4b%8b",rot,val)

op add(..., imm:immediat)
  syntax =  format (\add ...,%s",...,imm.syntax)
  image =  format (\... %s",...,imm.image)
 ...
```

Cet exemple, pris de [50], montre comment GLISS-nML permet de calculer la valeur exacte d'un opérande immédiat pour une instruction *ARM*.

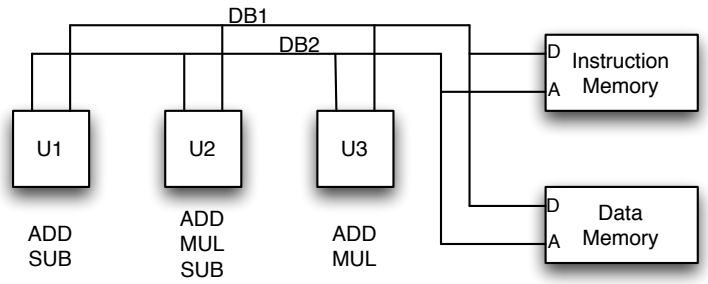
Enfin, GLISS-nML a été utilisé pour générer des simulateurs de jeux d'instructions pour des architectures très variées, comme *ARM*, *PowerPC*, *HCS12X* et *Sharc*.

### 2.3.2 ISDL

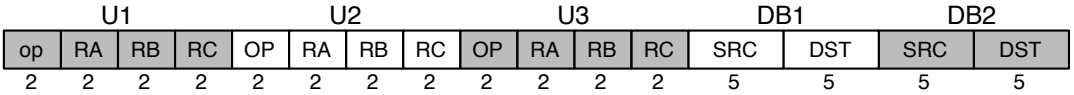
Le langage de description du jeu d'instructions [24] (ISDL, *Instruction Set Description Language*) a été developpé au *Massachusetts Institute of Technology* (*MIT*), Cambridge, États Unis. Comme nML, ISDL se présente aussi sous la forme d'une grammaire attribuée, il a été utilisé par le compilateur AVIV [27] et le générateur du simulateur Gensim [25].

ISDL est plus souple et sa sémantique est plus forte que celle de nML. Il permet la description d'une grande variété d'architectures, en mettant l'accent sur les architectures *VLIW* (*Very Long Instruction Word*). Comme dans nML, la description du jeu d'instructions contient des informations comportementales et structurelles. En outre, il permet la description de la configuration matérielle de la micro-architecture, qui est mélangée avec le comportement des instructions.

Pour expliquer l'organisation d'une description ISDL, considérons un exemple d'architecture VLIW pris de [24]. Comme le montre la figure 2.2(a), cette architecture contient 3 unités fonctionnelles (*U1*, *U2* et *U3*) contenant chacune 4 registres 8 bits, 2 mémoires (*Instruction Memory* capable de stocker jusqu'à 256 instructions 44 bits, et *Data Memory* de 32 entrées de 8 bits chacune) et 2 bus (*DB1* et *DB2*) assurant la liaison entre les deux mémoires et les bancs registres de chaque unité fonctionnelle.




(a)




(b)

FIGURE 2.2 – (a) Un exemple d'architecture *VLIW*. (b) Le mot instruction (44 bits) de cette architecture (le nombre de bits est indiqué en dessous de chaque élément).

Une description ISDL comporte six parties :

– la partie *format du mot instruction* : elle permet de décrire le mot instruction de l'architecture. La description doit être faite dans l'ordre partant du bit de poids fort (*MSB*). Le mot est divisé en un ou plusieurs champs contenant eux-mêmes un ou plusieurs sous-champs comme le montre l'exemple suivant (voir figure 2.2(b)) :

```
U1 = OP[2], RA[2], RB[2], RC[2];
U2 = OP[2], RA[2], RB[2], RC[2];
U3 = OP[2], RA[2], RB[2], RC[2];
DB1 = SRC[5], DEST[5];
```

```
DB2 = SRC[5], DEST[5];
```

– la partie *ressources de stockage* : elle permet de décrire la taille ainsi que la largeur des mémoires et des bancs registres. Pour les registres individuels, comme le compteur programme, la largeur du registre en bits est spécifiée. L'exemple suivant décrit les mémoires, les bancs registres et le compteur programme de l'architecture VLIW présentée plus haut (voir figure 2.2(a)).

```
Instruction Memory INST = 0x100 x 0x2C  //256 instructions
                                        //de 44 bits
Memory DM = 0x20 x 0x8   //Data Memory: 32 entrées
                         //de 8 bits
RegFile U1 = 0x4 x 0x8   //4 registres de 8 bits
RegFile U2 = 0x4 x 0x8
RegFile U3 = 0x4 x 0x8
ProgramCounter PC = 0x8  //8 bits pour PC
```

– la partie *définitions globales* : elle sert à définir des symboles (*Token*), des non-terminaux (*Non_Terminal*) et des fonctions de découpage (*Split functions*) qui sont utilisés dans les autres parties de la description.
Les symboles permettent de représenter des entités telles que les noms de registres, les noms de mémoires, et les constantes immédiates. Ils peuvent être utilisés pour regrouper syntaxiquement des entités liées telles que les noms des registres d'un banc registres. Pour différencier les éléments dans un groupe, un symbole retourne une valeur identifiant l'élément à représenter. Par exemple, les noms de registre tel que `U1_R0` à `U1_R3` peut être abrégé par un symbole `U1_R` dont la valeur retournée correspond au numéro de registre (0, 1, 2 ou 3) comme le montre l'exemple suivant :

```
//        syntaxe       symbole     valeur
Token "U1.R" [0..3]      U1_R     {[0..3];};
...
```

Les non-terminaux permettent de mutualiser les structures communes dans les définitions d'opérations (par exemple, les modes d'adressage). Prenons l'exemple suivant, le non-terminal `U1_RA` peut être utilisé dans les opérations comme paramètre pour référencer, par exemple, un registre source qui peut être soit `U1_R0`, `U1_R1`, `U1_R2` ou `U1_R3`.

```
//                syntaxe        valeur      action
Non_Terminal U1_RA: U1_R {$$ = U1_R;} {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RB: U1_R {$$ = U1_R;} {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RC: U1_R {$$ = U1_R;} {U1[U1_R]} {} {} {} ;
...
```

Les fonctions de découpage permettent de définir comment une longue constante (par exemple, une longue adresse mémoire ou des données immédiates) peuvent être répartis en plusieurs sous-champs composant le mot binaire d'instructions ;

– la partie *jeu d'instructions* permet de définir les différentes opérations utilisant les unités fonctionnelles, les mémoires et les bus de l'architecture à décrire. Pour l'architecture présentée plus haut, 3 champs (`U1f`, `U2f` et `U3f`) sont nécessaires pour décrire les différentes opérations (*ADD*, *SUB* et *MUL*) que les 3 unités fonctionnelles (*U1*, *U2* et *U3*) peuvent exécuter comme le montre l'exemple suivant :

```
Field U1f:
 U1_add U1_RA, U1_RB, U1_RC  //la syntaxe de l'opération add
                             //et ses paramètres
    //l'assignation des champs de bits (sert au décodage):
   {U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC;}
   {U1_RC <- ADD(U1_RA, U1_RB);}   //la sémantique
   { }  //s'il y a des effets secondaires, autre que
        //l'incrémentation du compteur programme (implicite)
   { Cycle = 1;  Size = 1;  Stall = 0;}   //les informations
   { Latency = 1;  Usage = 1;}            //temporelles
 ...
Field U2f:
 ...
Field U3f:
 ...
```

– la partie *contraintes* : elle utilise un ensemble de règles booléennes afin de définir, pour le compilateur, l'ensemble des opérations qui ne peuvent pas être exécutées en parallèle ;
– la partie *informations d'optimisation* permet au compilateur d'optimiser le code généré.

Nous pouvons remarquer que ISDL permet de générer un simulateur de jeu

d'instructions mais aussi un simulateur temporel en se basant sur les informations temporelles spécifiées dans les opérations mais ceci sans modélisation de la micro-architecture.

En général, les ADLs orientés jeu d'instructions permettent une description hiérarchique du jeux d'instructions basée sur une grammaire attribuée. Cette caractéristique donne la possibilité de mutualiser les parties communes entre les instructions en les groupant ensemble, permettant ainsi de simplifier la description. Par contre, il n'est pas possible de générer un simulateur précis au cycle près en raison d'absence de quelques détails structurels (informations temporelles et la possibilité de décrire le vrai fonctionnement d'un pipeline).

## 2.4 MIMOLA, un ADL structurel

MIMOLA [61, 4, 40] est un langage basé sur une approche orientée structure interne. Il a été élaboré par Gerhard Zimmermann et un groupe de chercheurs à *radio astronomy observatory* de l'Université de Kiel, Allemagne.

MIMOLA est l'un des premiers langages conçu spécialement pour la synthèse de haut niveau des processeurs et non seulement pour la simulation (*hardware-software co-design*). Il permet de décrire la structure matérielle sous forme d'un réseau de connexion (en anglais, *netlist*) constitué d'un ensemble de modules et d'un schéma d'interconnexion détaillé comme le montre l'exemple suivant [4], pris de [40], décrivant une simple architecture nommée `simple_hardware` :

```
module simple_hardware;  (* tête du module *)
  structure
    type word = (15:0);  (* vecteur de bits *)
    module Srom( adr addr:word; out f:word);  (* types de module *)
      var code::array [0..#FFFF] of word;
      begin
       f<-code[addr]  after 10;  (* 10 est un delai *)
      end;
    module BAlu1 ...
    module BAlu2 ...
    module Regi ...
    module Sram ...
    module Rflipflop ...
    parts
     I : Srom;  (* instances du module *)
     pc,reg : Regi;
```

4. Les commentaires sont donnés sous la forme *(*commentaire*)*.

```
    Mem : Sram;
    cc : Rflipflop;
  connections  (* interconnexion des instances du module *)
    pc.f -> I.addr;  (* connexion de la sortie pc.f à l'entrée
                        adresse I.addr de la ROM *)
    ...
end_structure;
```

Dans une description MIMOLA, le comportement à mettre en œuvre dans un module est capturé dans la partie définition du programme. Pour faciliter le plus possible l'utilisation de MIMOLA, la description de la partie programme est très similaire au langage de programmation PASCAL, très populaire à l'époque. L'utilisateur a le choix d'écrire cette partie au niveau application ou bien la représenter sous forme d'un interpréteur d'instructions pour un jeu d'instructions donné. Dans le premier cas, typiquement aucun jeu d'instructions ne sera obtenu dans la conception finale. Alors que dans le second cas, la synthèse va générer un interpréteur micro-codé pour le jeu d'instructions.

Ce langage offre l'avantage que la même description est utilisée à la fois pour la synthèse du processeur et la génération de code. Le jeu d'instructions est extrait de la partie programme, cette tâche peut être difficile pour des instructions complexes. En règle générale, MIMOLA est considéré comme un langage très bas niveau et est laborieux à écrire et à modifier. En outre, la simulation est lente.

## 2.5 Les ADLs mixtes

Parmi les ADLs mixtes les plus connus, nous citerons LISA [62, 46], et EXPRESSION[26].

### 2.5.1 LISA

LISA [62, 46, 40, 11] (*Language for Instruction-Set Architecture*) est un langage qui a été élaboré à l'Université d'Aix la Chapelle, à l'Institut *ISS* (pour *Integrated Signal Processing Systems*), Allemagne. C'est actuellement la société *Synopsys*[5] qui est chargée du développement, après le rachat de *LisaTek*. LISA permet la génération automatique d'un nombre conséquent d'outils à partir d'une description de l'architecture matérielle : compilateur C, Assembleur, éditeur de liens et simulateur pour la phase de conception (exploration), mais aussi la production d'un

5. http://www.synopsys.com/Tools/SLD/ProcessorDev/Pages/default.aspx

code *VHDL* pour l'implémentation. Toutes les informations sont réunies dans une même description pour éviter les problèmes de cohérence entre les différents outils.

LISA a été développé à travers 6 modèles :

- le *modèle mémoire* contient les informations pour permettre de décrire à la fois les zones mémoires, et les registres ;
- le *modèle de ressources* permet de lister les ressources matérielles disponibles ainsi que les possibilités d'utilisation des ressources. Par exemple, la capacité d'un registre à être accessible en lecture/écriture ;
- le *modèle comportemental* permet de décrire une activité matérielle sous la forme d'une machine à états. L'exécution d'une instruction change l'état du système ;
- le *modèle du jeu d'instructions* permet de vérifier la validité d'une instruction, par exemple, la compatibilité entre le mode d'adressage et l'instruction associée ;
- le *modèle temporel* définit les séquences entre les instructions et le temps d'attente entre les enchaînements. Il donne aussi le temps d'exécution des instructions ;
- le *modèle de la micro-architecture* permet de regrouper les fonctionnalités dans une seule entité (par exemple l'addition et la soustraction dans l'unité arithmétique et logique).

Pour la génération des simulateurs, les *modèles de ressources* et *de micro-architecture* ne sont pas utilisés (ils le sont pour la génération du code *VHDL* notamment).

LISA permet une description formelle des architectures programmables, leurs interfaces et leurs périphériques. C'est un ADL souple, il facilite la description des différents processeurs. Dans de nombreux aspects, LISA contient des idées qui sont analogues à nML et sa syntaxe est très similaire au langage de programmation C, surtout pour la description du comportement du jeu d'instructions qui peut être spécifiée directement en C dans la section `BEHAVIOR`.

En utilisant LISA, la description d'une architecture est formée des deux déclarations principales suivantes : la définition des ressources et la définition des opérations. La figure 2.3 montre comment à partir de ces deux déclarations, les 6 modèles cités plus haut sont obtenus.

#### 2.5.1.1 Définition des ressources

La définition des ressources décrit les unités de stockage de l'architecture matérielle (comme les registres, la mémoire et le pipeline). Ces ressources permettent de mémoriser l'état de l'architecture programmable sous forme de valeurs et un tag indiquant la disponibilité limitée des ressources pour effectuer les différentes opérations. La déclaration des ressources se fait dans la section

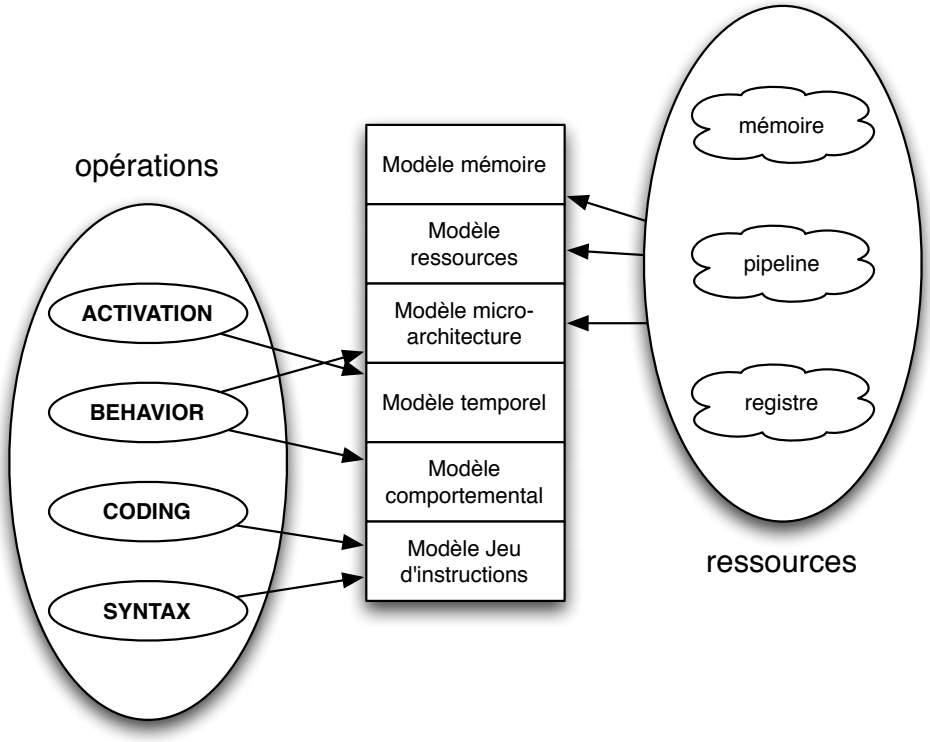



FIGURE 2.3 – La composition générale d'une description de processeur en utilisant LISA. Cette description permet de générer les 6 modèles de LISA.

`RESOURCE`, comme le montre l'exemple suivant :

```
RESOURCE {
  PROGRAM_COUNTER int pc;
  CONTROL_REGISTER int instruction_register;
  REGISTER char mreg[0..15];
  MEMORY int prog_mem {
    BLOCKSIZE (32 , 32) ;
    SIZE (0x10000) ;
    FLAGS (R | W | X);
 } ;
  MEMORY int data_mem {
    BLOCKSIZE (32 , 32) ;
    SIZE (0x4000) ;
    FLAGS (R | W);
```

```
  } ;
}
```

Cet exemple décrit la section `RESOURCE` d'un simple processeur non pipeliné. Dans le cas d'un processeur pipeliné, il faut ajouter dans cette section (`RESOURCE`) la description du pipeline qui se fait en utilisant le mot clé `PIPELINE`. Les étages du pipeline ainsi que les éléments de stockage entre deux étages du pipeline sont également définis comme le montre l'exemple décrivant le pipeline de 4 étages (Fetch, Decode, Execute et Write Back) suivant :

```
PIPELINE pipe = {F,D,E,WB};
PIPELINE_REGISTER IN pipe {
  int inst_register;
  PROGRAM_COUNTER short prog_count;
  REGISTER bit [24] src1, src2, dest;
}
```

De plus, il faut décrire les contraintes d'accès aux ressources qui doivent être toujours décrites dans la section `RESOURCE`. Par exemple, supposons que le banc de registres `b` est en mesure d'effectuer, en parallèle, 3 accès en lecture et 2 accès en écriture, cette contrainte est décrite de la manière suivante :

```
U32 b[0..15] { PORT ( READ = 3 OR WRITE = 2) ; };
```

#### 2.5.1.2 Définition des opérations

La déclaration des opérations permet de décrire le jeu d'instructions du processeur (la syntaxe, le code binaire et le comportement des instructions) dans une structure arborescente, ainsi que le modèle temporel (exécution des instructions à travers les étages du pipeline décrits dans la partie déclarations des ressources). La déclaration des opérations se fait à travers 6 sections :

- La section `CODING` : elle décrit le code binaire des instructions, elle sert pour la phase de décodage ;
- La section `SYNTAX` : elle permet d'attribuer une syntaxe textuelle à une instruction (pour permettre, par exemple, le désassemblage) ;
- Les sections `BEHAVIOR` et `EXPRESSION` : elles permettent de décrire le comportement des instructions. La section `BEHAVIOR` est basée sur le langage de programmation C, elle peut contenir des variables locales ainsi que des appels des fonctions qui sont, soit écrit directement en C (fonctions externes), soit des opérations interne de LISA. Les ressources décrites dans la section `RESSOURCE` peuvent être accédées également dans cette section. La section `EXPRESSION` définit un objet qui sera consulté par la section `BEHAVIOR`, elle

est surtout utilisé pour les opérandes et les autres accès aux ressources (de ce fait, elle n'apparaît pas sur la figure 2.3) ;
- La section `ACTIVATION` : permet d'ordonnancer les opérations pour les exécuter dans le pipeline défini dans la section `RESOURCE` ;
- La section `DECLARE` : cette section contient les déclarations locales ainsi que les appels des autres opérations LISA (de ce fait, elle n'apparaît pas sur la figure 2.3) ;

L'exemple suivant décrit l'instruction `ADD` qui fait l'addition du contenu des deux registres sources et stockent le résultat dans le registre destination.

```
OPERATION ADD {
  DECLARE {
   GROUP src1, src2, dest = {register} ;  //permet de mutualiser
                                          //les parties communes
                                          //entre plusieurs objets
  }
  CODING { 0b101011 src1 src2 dest }
  SYNTAX { "ADD" dest "," src1 "," src2 }
  BEHAVIOR { dest = src1 + src2 ; }
}

OPERATION register {
  DECLARE { LABEL index ; }  //déclaration d'une variable locale
  CODING { index=0bx[4] }    //index du registre entre 0..15
                             //(codé sur 4 bits)
  SYNTAX { "A" index }       //nom du registre est A0..A15
  EXPRESSION { R[index] }    //retourne l'identifiant pour
                             //l'accès registre
}
```

Si un pipeline a été défini dans la section `RESOURCE`, toutes les opérations doivent être assignées explicitement à un étage particulier de pipeline. Par exemple, pour l'instruction `ADD` décrit ci-dessus, ceci est fait de la manière suivante :

```
OPERATION fetch IN pipe.F {
  DECLARE { INSTANCE decode ; }  //référence à l'opération decode
  BEHAVIOR { inst_register = prog_mem[pc] ; }
  ACTIVATION { decode }
}

OPERATION decode IN pipe.D {
```

```
  DECLARE { INSTANCE add, writeback ; }
  ACTIVATION { add, writeback }
}

OPERATION add IN pipe.E {
  DECLARE { GROUP  src1, src2, dest = {register} ; }
  BEHAVIOR { result = src1 + src2 ; }
}

OPERATION writeback IN pipe.WB {
  DECLARE { REFERENCE  dest ; }  //fait référence au GROUP décrit
                                 //dans une opération supérieure
                                 //(ici add)
  BEHAVIOR { dest = result ; }
}
```

Dans LISA , la détection et la résolution des différents types d'aléas (structurel, de contrôle et de données) sont assurées par l'utilisation du concept de diagramme de Gantt étendu [62]. Le court-circuit (le fait de pouvoir envoyer un résultat aux instructions qui en ont besoin plus tôt dans un pipeline, c'est-à-dire avant qu'il ne soit disponible dans le registre destination) ainsi que l'ajout des suspensions pour bloquer le pipeline et le vidage du pipeline (pour résoudre les aléas de contrôle par exemple) peuvent aussi être décrits. Par contre, il nous semble que LISA ne permet pas de décrire les instructions de taille variables.

### 2.5.2 EXPRESSION

EXPRESSION [26, 40] est un ADL développé par le *Departement of Information and Computer Science* de l'Université de Californie (Irvine). Cet ADL est très fortement orienté vers la phase de conception (exploration d'architecture), mais la génération de simulateurs et de compilateurs a aussi été abordée [26, 23].

Comme dans le cas de LISA, la description dans EXPRESSION est formée des deux sections principales suivantes :

- la section *behavior* qui est formée de 3 sous-sections : la spécification des opérations qui décrit le jeu d'instructions du processeur, la description des instructions qui capture le parallélisme offert par l'architecture (une instruction est vue comme un ensemble d'opérations qui peuvent être exécutées en parallèle) et le mappage des opérations qui permet au compilateur d'effectuer plusieurs optimisations, comme le montre l'exemple suivant [40] :

```
#spécification des opérations qui utilisent l'ALU
```

```
( opgroup aluOps (add,sub,...))
( opcode add
  ( operands (s1 reg) (s2 reg) (dest reg))  #sources et
                                #destination de type reg
  ( behavior dst = s1 + s2)
  ( format 000101 dst(25-21) s1(20-16) s2(15-0))
)
...
( vargroup  #définit le type des opérandes, ici 'reg'
            #fait référence au banc de registres
  (reg RegisterFile)
  ...
)

 #description des instructions
( INSTR (SLOTS (UNIT ALU) (UNIT MULT) ... ))

 #mappage des opérations
( OP_MAPPING
 #multiplication de x par 2 peut être remplacée par add x x
  (( GENERIC (mult src1 #2 dst))
  ( TARGET (add src1 src1 dst)))
  ...
)
```

– la section *structure* de l'architecture qui peut être vue comme un réseau de connexion où les composants sont les nœuds et les interconnexions les terminaisons. Elle est composée de 3 sous-sections :
  – la spécification des composants qui permet de décrire les composants matériels (les unités fonctionnelles, les ports, les connexions et les bus) et de donner des informations temporelles sur les unités pipelinées ou multicycles ;
  – la description du pipeline et du chemin de transfert de données qui permettent respectivement de décrire le flux d'instructions à travers les différents étages du pipeline et fournir un mécanisme permettant de spécifier les transferts de données valides ;
  – la description de la mémoire qui permet de décrire les types et les attributs des différents composants de mémorisation (mémoire cache, les registres, SRAMs,...).

  L'exemple suivant [40] montre comment se fait la description de la section

*structure* d'une architecture.

```
 #spécification des composants
( SUBTYPE UNIT FetchUnit DecodeUnit ExecUnit ...)
( SUBTYPE PORT UnitPort Port ...)
( SUBTYPE CONNETION MemoryConnection RegConnection ...)
( SUBTYPE LATCH PipelineLatch MemoryLatch)
(FetchUnit Fetch
  ( capacity 3)       #jusqu'à 3 accès en parallèle
  ( timing (all 1))  #1 cycle d'horloge par instruction
  ( opcodes all       #toutes les instructions
  ...
) ...
(ExecUnit ALU
  ( capacity 1) ( timing (add 1) (sub 1) ...)
  ( opcodes add sub ...)  ...
)...

 #pipeline et chemin de transfert de données
( pipeline Fetch Decode Execute WriteBack)
(Execute ( parallel ALU loadStore ...))
(loadStore ( pipeline addrCalc MemCntrl))
( dtpaths (writeBack RegisterFile) (L1Data L2) ...)

 #spécification de la section mémorisation
( DCache L1Data
  ( wordsize 64) ( linesize 8) ( associativity 2)
  ( accessTime 1) ...
)
( DCache L1Inst
  ( wordsize 64) ( linesize 8) ( associativity 2)
  ( accessTime 1) ...
)
( DCache L2
  ( wordsize 64) ( linesize 16) ( associativity 4)
  ( accessTime 10) ...
)
...
)
```

Un des points forts de l'ADL EXPRESSION est que les tables de réservation, nécessaires pour tester les contraintes sur l'utilisation des ressources (aléas structurels) par les opérations dont les cycles d'exécution se chevauchent, sont générées automatiquement à partir des informations structurelles [22]. En effet, chaque instruction est exécutée à travers les étages du pipeline tout en accédant aux composants matériels à travers des chemins de transfert de données ; donc il est possible de tracer le chemin d'exécution des instructions pour pouvoir générer les tables de réservation précisément. Cette approche libère l'utilisateur de cette tâche (description des tables de réservation pour chaque opération) qui est souvent laborieuse.

Comme nous l'avons remarqué, dans EXPRESSION, l'utilisateur a la charge de spécifier les interconnexions entre les différentes entités. Les concepteurs d'EXPRESSION justifient ce choix par leur volonté de produire un simulateur précis et un compilateur à partir de la description réalisée avec leur ADL. Le niveau d'abstraction auquel travaille EXPRESSION se trouve juste au dessus du niveau des portes logiques. Dans la description, le grain du modèle permet d'aller décrire les bascules (latches) entre les différentes éléments. De plus, comme dans LISA, la description d'un jeu d'instructions de taille variable n'est pas possible avec EXPRESSION.

## 2.6 Positionnement des travaux par rapport à l'état de l'art

Par rapport à la figure 2.1 déjà présentée en début de chapitre, notre langage HARMLESS se situe avec les ADLs mixtes comme le montre la figure 2.4.

Contrairement à d'autres langages de description d'architecture matérielle (comme MIMOLA, EXPRESSION, ...), l'objectif premier du langage HARMLESS n'est pas d'aider à la conception de processeurs, et n'est donc pas de permettre de raffiner la description jusqu'à permettre de synthétiser un prototype matériel dans un langage de description matériel (VHDL ou Verilog par exemple). Nos objectifs globaux sont donc d'obtenir, à partir d'une description du processeur, un simulateur *ISS* et un simulateur *CAS*.

Nos travaux se rapprochent le plus de ceux effectués pour LISA, mais un ensemble de différences sont mentionnées ci-après, en ce qui concerne la modélisation d'un pipeline.

L'ADL Lisa (voir section 2.5.1) permet de décrire un pipeline d'une manière abstraite (par exemple, le concepteur n'a pas besoin de donner la structure du processeur) et, de ce point de vue, est proche de HARMLESS. Toutefois, HARMLESS utilise une approche différente. Dans les deux ADLs, le concepteur définit les étages du pipeline. Avec LISA, les registres du pipeline sont explicitement définis

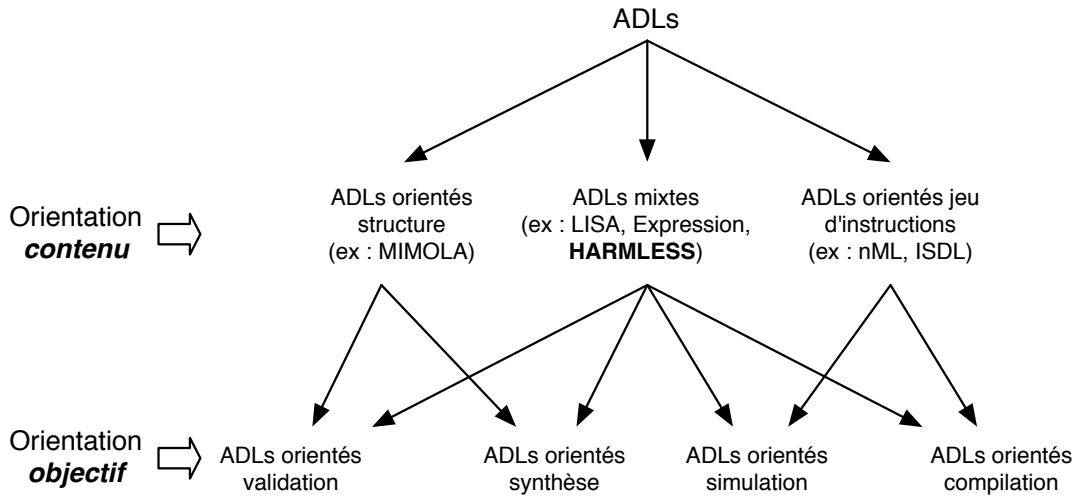



FIGURE 2.4 – Classification des ADLs selon les deux aspects : *contenu* et *objectif*. HARMLESS est un *ADL mixte*.

et toutes les instructions doivent être assignées explicitement à un étage particulier de pipeline. En conséquence, un changement dans la description du pipeline nécessite la réécriture de la description du jeu d'instructions. Avec HARMLESS, le concepteur assigne les composants à des étages du pipeline (voir chapitre 6). Les composants et la description du jeu d'instructions sont indépendants de la description du pipeline et un changement dans le pipeline ne conduit pas à une réécriture de la description du jeu d'instructions ou la description du composant.

## Autres caractéristiques d'HARMLESS

HARMLESS permet une description :
- incrémentale ;
- indépendante de la micro-architecture ;
- concise ;
- permettant la vérification.

### Une description incrémentale

Contrairement aux autres ADLs présentés dans les sections 2.3, 2.4 et 2.5, et qui décrivent le format binaire, la syntaxe textuelle et le comportement des instructions dans une seule vue, nous distinguons 3 vues pour la description d'un jeu d'instructions :

- la vue *binaire*, c'est-à-dire le codage des instructions. Cette description permettra de construire un décodeur qui, à partir d'un code exécutable, construira une représentation interne des instructions de ce programme dans le simulateur ;
- la vue *comportementale*, c'est-à-dire les opérations que les instructions effectuent lorsqu'elles s'exécutent ;
- la vue *syntaxique*, c'est-à-dire la syntaxe textuelle associée à une instruction.

Le langage doit autoriser une description incomplète du jeu d'instructions. Par exemple, il n'est pas forcément nécessaire de disposer de la vue syntaxique des instructions pour engendrer un décodeur et un simulateur.

### Une description indépendante de la micro-architecture du processeur

Cette contrainte est un point important pour trois raisons.

Tout d'abord, dans la phase de mise au point du modèle de processeur, il est souhaitable de ne pas avoir à décrire les aspects temporels du modèle (c'est-à-dire, le pipeline et les concurrences d'accès aux composants matériels qui nous permettent de déduire les caractéristiques temporelles).

Ensuite, cette indépendance permet d'engendrer deux simulateurs : un simulateur du jeu d'instructions qui ne prend pas en compte le comportement temporel et un simulateur précis au cycle près (prenant alors en compte la micro-architecture du processeur). Dans notre réalisation, ces deux simulateurs peuvent être engendrés séparément ou conjointement. Dans le second cas, les deux simulateurs partagent le même contexte d'exécution et il est possible de commuter de l'un à l'autre au cours de la simulation. De cette manière, il est possible de simuler rapidement les sections de programme où la précision des temps d'exécution n'est pas nécessaire (comme par exemple les initialisations) et de basculer sur la simulation précise au cycle près dans les autres cas.

Enfin, elle permet de mutualiser la description du jeu d'instructions entre plusieurs micro-architectures d'une famille de processeurs.

### Une description concise

La description du jeu d'instructions doit être la plus concise possible afin de réduire le temps nécessaire à la description d'un processeur et les risques d'erreurs.

Trois choix de conception permettent cette concision :

- le langage fournit des opérateurs permettant de manipuler les champs de bits : extraction, concaténation, masquage, etc ;
- les tailles de données sont spécifiées au bit près ;
- le langage permet de mutualiser les parties communes des instructions selon les différentes vues.

**Une description permettant la vérification**

Afin d'assurer que le code engendré est, autant que faire se peut, exempt d'erreurs, le langage offre les caractéristiques suivantes :

- *Typage fort des variables :* cette contrainte permet de vérifier que les tailles des opérandes et des résultats de calculs sont compatibles au bit près entre eux, et avec les variables employées dans les expressions ;
- *Pas de possibilité d'inclure des sections en langage C :* certains langages permettent, dans la description, d'inclure des sections directement en langage C. Cette solution, plus simple pour l'analyse syntaxique et la génération de code, a été écartée afin de pouvoir vérifier entièrement la cohérence de la description.

## 2.7 Conclusion

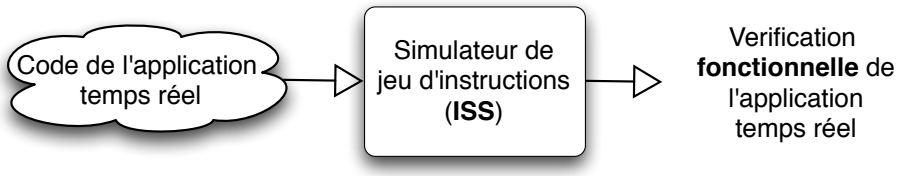
Dans ce chapitre, nous avons présenté plusieurs ADLs existants : des ADLs orientés jeu d'instructions (nML, Gliss-nML et ISDL), un ADL orienté structure interne (MIMOLA) et des ADLs mixtes (LISA et EXPRESSION) tout en positionnant nos travaux par rapport à ces ADLs.

En effet, HARMLESS est un *ADL mixte* permettant une description concise des différentes parties d'un processeur : le jeu d'instructions et la micro-architecture (les composants matériels, le pipeline et les concurrences d'accès aux différents composants matériels). L'originalité de *HARMLESS*, vis à vis des autres ADLs comme LISA, est le découplage de la description du jeu d'instructions de la description de la micro-architecture. L'une des conséquences est de permettre la génération des deux types de simulateurs indépendamment et simultanément. Une autre conséquence est une construction incrémentale de la description : 4 vues séparées permettent de décrire d'une part le jeu d'instructions (3 vues pour la syntaxe, le format binaire et la sémantique) et d'autre part la micro-architecture du processeur. Ceci facilite la réutilisation du code sur une nouvelle architecture cible.

# Première partie

# Simulateur de jeu d'instructions (*ISS*)

Les simulateurs de jeu d'instructions (*ISS*) sont communément utilisés dans les processus de développement des systèmes embarqués pour la validation fonctionnelle primaire de code et pour l'exploration d'une nouvelle conception de jeu d'instructions. Un tel simulateur (voir la figure ci-dessous) accepte en entrée le code de l'application temps réel, et l'exécute. En sortie, l'examen des résultats produits permet de vérifier la bonne exécution fonctionnelle de cette application.




Dans cette partie, nous allons présenter la génération automatique du simulateur de jeu d'instructions en utilisant notre langage de description d'architecture matérielle HARMLESS. Il diffère d'autres langages dans plusieurs aspects : il permet plus d'expressivité dans la description et offre naturellement une description flexible et incrémentale en séparant explicitement la syntaxe (mnémonique), le format (le code binaire) et la sémantique (comportement fonctionnel des instructions). L'aspect « incrémental » est également important car il permet d'obtenir par exemple le désassembleur en fournissant le format et la description de la syntaxe des instructions. Dans un deuxième temps, l'ajout de la description de la sémantique permet d'obtenir le simulateur.

Cette partie est formée de deux chapitres. Dans le premier, nous allons parler de la description d'une architecture fonctionnelle dans HARMLESS alors que dans le second, nous allons présenter comment nous engendrons le simulateur *ISS* tout en mettant en évidence les avantages cités ci-dessus qu'offrent notre langage HARMLESS. Dans cette partie, ma contribution porte surtout sur la description de la vue syntaxique qui permet la génération des mnémoniques des instructions, les deux autres vues étaient déjà globalement faites par Mikaël Briday.

# Chapitre 3

# Description d'une architecture fonctionnelle

## 3.1 Introduction

Dans ce chapitre, nous nous intéressons au simulateur *ISS* qui prend en compte le comportement des instructions indépendamment du temps nécessaire pour leur exécution. Dans ce type de simulateur, l'architecture interne du processeur n'est pas modélisée et seul le comportement fonctionnel est pris en compte avec pour avantage une plus grande rapidité. La chaîne de développement associée au langage de description d'architecture HARMLESS, pour obtenir automatiquement le simulateur *ISS* est donnée sur le schéma 3.1.

En effet, pour pouvoir engendrer automatiquement un tel simulateur, il faut disposer d'une :

- description du jeu d'instructions qui est l'ensemble des opérations élémentaires qu'un processeur est capable d'exécuter ;
- description fonctionnelle des composants matériels (ALU pour Arithmetic Logic Unit, mémoire, registres, ...) qui sont nécessaires pour simuler le comportement des instructions (addition, accès mémoire, lecture/écriture registre, etc).

La description fonctionnelle des composants matériels peut être mise directement à l'intérieur de la description du comportement des instructions. Mais, puisque notre but est de pouvoir générer en plus un simulateur précis au cycle près, avoir les deux descriptions séparées nous permettra d'associer des contraintes temporelles à l'utilisation des ressources matérielles par les instructions (voir chapitre 6).

Dans HARMLESS, l'organisation de la description d'une architecture fonctionnelle, syntaxiquement décrite en *BNF étendu* [30], suit le schéma général suivant :

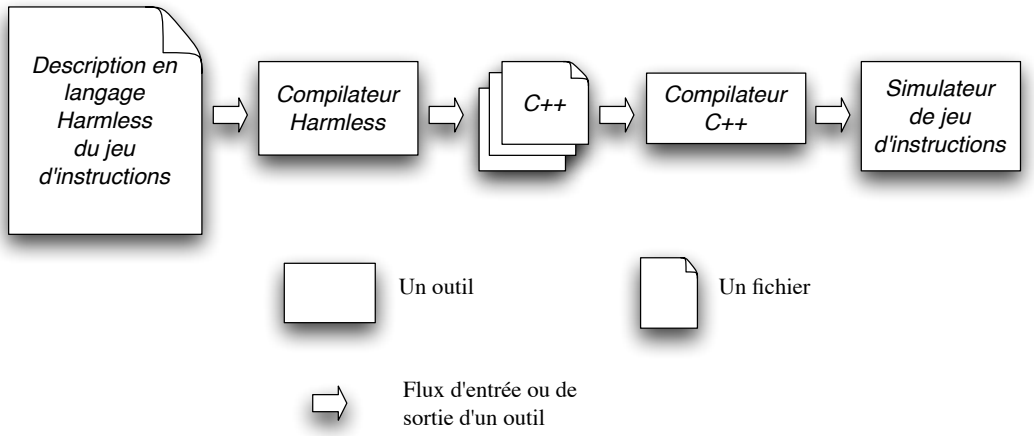



FIGURE 3.1 – Chaîne de développement du simulateur *ISS*.

```
<modelDeclaration>
{
  <inModel> |
  <default> |
  <component> |
  <format> |
  <behavior> |
  <syntax> |
  <printNumberType>
};
```

Il est possible de mettre chacune de ces règles, dans n'importe quel ordre, autant de fois que nécessaire (même 0).

Les différentes règles sont :

- `modelDeclaration` c'est la déclaration du modèle du processeur. Il est possible de déclarer plusieurs modèles dans la même description, voir section 3.2.1.1 ;
- `inModel` cette règle sert à la description de plusieurs modèles, voir section 3.2.1.1 ;
- `default` cette règle permet de définir les paramètres par défaut du modèle. Elle est obligatoire, et définie dans la section 3.2.1.3 ;
- `component` cette règle définit un *composant*. Elle est expliquée dans la section

3.3 ;
- `format`, `behavior` et `syntax` permettent de décrire les 3 vues du jeu d'instructions. Elles sont définies dans les sections 3.2.1.3, 3.2.1.5 et 3.2.1.4 ;
- `printNumberType` est utilisée pour la description de la syntaxe, voir section 3.2.1.4 ;

Dans un premier temps, nous verrons comment HARMLESS permet une description incrémentale en séparant la description du jeu d'instructions en trois vues : binaire, syntaxique et sémantique ; ensuite nous aborderons la description fonctionnelle des composants matériels, tout en se servant des exemples extraits de la description du *HCS12* de *Freescale*, de son co-processeur *XGate* ainsi que de l'*AVR* de l'*ATMEL*.

## 3.2 Description du jeu d'instructions

Comme nous l'avons déjà précisé, le jeu d'instructions est l'ensemble des opérations qu'un processeur peut faire. Un code, dit code-opération, spécifie chaque opération élémentaire et des opérandes peuvent être associés à ce code. Donc une instruction est formée en général de deux champs : un pour le code et un autre pour les opérandes. Nous pouvons classer les instructions en plusieurs catégories dont les principales sont :
- opérations logiques et arithmétiques : opérations logiques telles que OU, ET, NON, OU exclusif,... et arithmétiques telles que addition, soustraction, multiplication ou division ;
- accès à la mémoire : chargement/sauvegarde en mémoire et transferts de données entre registres ;
- contrôle : comme les branchements conditionnels et inconditionnels, etc.

### 3.2.1 Les trois vues

Comme indiqué précédemment, HARMLESS utilise 3 vues pour décrire le jeu d'instructions d'un processeur, ce qui rend la description souple et plus facile à modifier ou adapter à un autre processeur ayant un jeu d'instructions proche. Les 3 vues sont les suivantes :
- la vue binaire (*format*) est utilisée pour décrire le format binaire des instructions, afin de permettre l'étape de décodage des instructions ;
- la vue comportementale (*behavior*) permet de décrire le comportement des instructions, afin de les simuler ;
- la vue syntaxique (*syntax*) permet de décrire le mnémonique de l'instruction, afin de générer le désassembleur.

#### 3.2.1.1 Généralités

**Une modélisation arborescente** Chaque vue est un ensemble d'arbres où un nœud décrit un morceau de format, de sémantique ou de syntaxe. Une description de nœud est conforme à la syntaxe suivante :

```
1 <type> <name> [<type_options>]
2   <description>
3 end <type>
```

où `<type>` peut être remplacé par `format`, `syntax` ou `behavior`, suivi d'un nom (`<name>`) donné par l'utilisateur, suivi parfois par des options dépendantes de la vue en question, et enfin vient la partie `<description>`. Le mot clé `end` termine la spécification du `<type>`.

Comme dans un langage de spécification de grammaire, HARMLESS permet de décrire, pour chaque vue, si un nœud non-terminal est construit en agrégeant des sous-nœuds ou en choisissant un nœud, ou un ensemble de nœuds, parmi plusieurs. Par défaut, un nœud non terminal agrège les sous-nœuds. Le constructeur `select` permet de choisir un sous-nœud parmi plusieurs (ou un ensemble de sous-nœuds parmi plusieurs). En utilisant cette méthode, dans chaque vue, une instruction est représentée par une branche dans un arbre. Les instructions partageant une partie commune dans une vue partagent des nœuds dans les racines de l'arbre, alors que les parties spécifiques se trouvent dans les feuilles de l'arbre.

**Quelques caractéristiques de HARMLESS** C'est un langage fortement typé ; il offre les types de données signé et non signé et la taille est définie au bit près, ce qui permet de supprimer, pendant la compilation, le plus grand nombre possible d'erreurs en vérifiant par exemple que les tailles des opérandes et des résultats de calculs sont compatibles entre eux. Actuellement, HARMLESS ne peut gérer que les données de type entier. D'autres part, les nombres entiers peuvent être écrits dans différentes bases : binaire, décimale, octale et hexadécimale, en précédant le nombre par respectivement : `\b`, `\d`, `\o` et `\x`. Par défaut, c'est le format décimal qui est utilisé :

```
u17 val1 -- val1 est definie sur 17 bits, non signee
s9 val2  -- val2 est definie sur 9 bits, signee
u1 bool  -- definition d'un type sur 1 seul bit

38       -- 38 en decimal
\d12     -- 12 en decimal -> defaut
\b100    -- 4 en binaire
\o70     -- 56 en octal
\x2F     -- 47 en hexadecimal
```

HARMLESS offre plusieurs autres caractéristiques importantes qui permettent de faciliter la description et la génération automatique du simulateur ISS. Par exemple, la somme de deux mots de $n$ bits chacun produit un résultat de $n+1$ bits. Avec cela, l'implémentation de la retenue (« carry » en anglais) et du dépassement (« overflow » en anglais) est plus facile. Les opérateurs pour extraire et concaténer les champs de bits sont aussi fournis :

```
  u16 val1 := \x5500
  u16 val2 := \x0055
  u17 result := val1+val2  -- resultat sur 17 bits
  u1 carry := result{16}   -- seulement le bit de poids fort
  u16 valResult := result{15..0}
```

En outre, dans HARMLESS, Il est possible de décrire plusieurs variantes de modèles. Ceci est notamment intéressant pour mutualiser une description pour les différentes variantes d'une architecture, permettant ainsi d'engendrer un simulateur par modèle. Dans ce cas, il faut préciser au début de la description que nous décrivons plusieurs modèles de la façon suivante :

```
model mod1, mod2, mod3
```

Dans cet exemple, la description sera faite pour les 3 modèles `mod1`, `mod2` et `mod3`. Par défaut, toute la description est commune à tous les modèles. Pour spécifier une partie (comme un composant, un `behavior`, un `format`, etc...) spécifique à un modèle donné, nous utilisons le constructeur `in` :

```
1 -- dans ce cas, la description entre {} sera valide
2 in mod1, mod2 { -- uniquement pour les modeles mod1 et mod2
3   ...
4 }
```

Il est également possible d'utiliser le joker `*`, ainsi que le mot clé `except` pour au contraire enlever des modèles :

```
1 in * except mod3 { -- identique a la description precedente
2   ...
3 }
```

Avant de détailler les 3 vues de notre langage HARMLESS, nous allons présenter le concept qui permet de relier ces différentes vues et que nous appelons *Signature d'une instruction.*

#### 3.2.1.2 Signature d'une instruction

Comme nous l'avons déjà dit, chaque vue est formée d'un ensemble d'arbres où une branche représente une instruction. Dans HARMLESS, un nœud dans un arbre peut avoir une ou plusieurs étiquettes, chacune définie par le caractère `#` suivi d'un identificateur.

Un ensemble d'étiquettes tout au long d'une branche d'un arbre forme l'identificateur *unique* d'une instruction et est appelé la *signature de l'instruction.* Cette signature permet de faire le lien entre les différentes vues (binaire, syntaxique et sémantique) comme montre la figure 3.2. En effet, pour une instruction donnée, le même ensemble d'étiquettes, formant la signature, sera présent dans les trois vues (à titre d'exemple, dans l'annexe A, pour l'instruction *OR*, nous pourrons remarquer que l'ensemble d'étiquettes `#triadicInst` et `#OR` est commun aux trois vues). Toutes les instructions partageant une même étiquette partagent des caractéristiques identiques (binaires, syntaxiques ou sémantiques).

Comme la signature est un ensemble d'étiquettes, ceci implique qu'il n'y a pas de notion d'ordre, et qu'il n'y a pas de comptage du nombre d'occurrence de chaque étiquette.

Dans certains cas, il est utile de pouvoir marquer des étiquettes car un même nœud peut être utilisé dans des contextes différents, par exemple lors de la lecture de 2 registres source d'une instruction (voir l'exemple suivant). Dans ce cas, une post-étiquette peut être utilisée et elle est représentée par le caractère `@` suivi d'un identificateur.

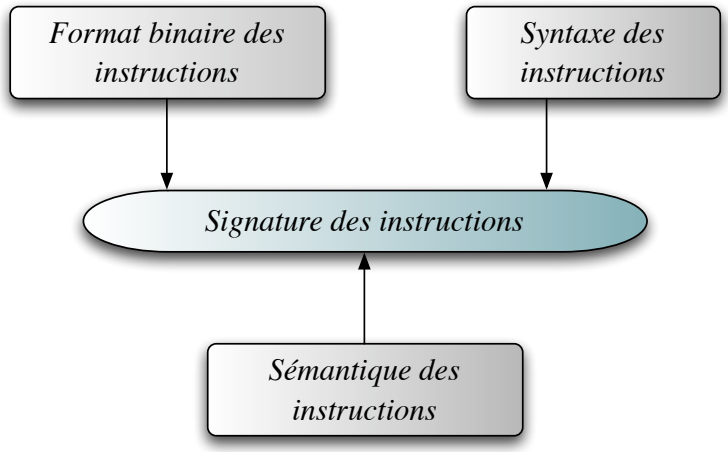



FIGURE 3.2 – Le lien entre les différentes vues : la « signature des instructions ».

Pour mettre en évidence la structure arborescente dans notre langage HARMLESS ainsi que l'utilisation des étiquettes et des post-étiquettes qui forment la signature d'une instruction, prenons par exemple le code (simplifié) suivant :

```
1  behavior readGPR
2    #read8
3    ...
4  end behavior
5
6  behavior writeGPR
7    #write8
8    ...
9  end behavior
10
11 behavior twoRegsOp
12   readGPR@src1    -- lit le premier registre source
13   readGPR@src2    -- lit le deuxieme registre source
14   select          -- instruction qui utilise 2 registres source
15     case #ADC   ...
16     case #ADD   ...
17     case #SUB   ...
18   end select
19   writeGPR
20 end behavior
```

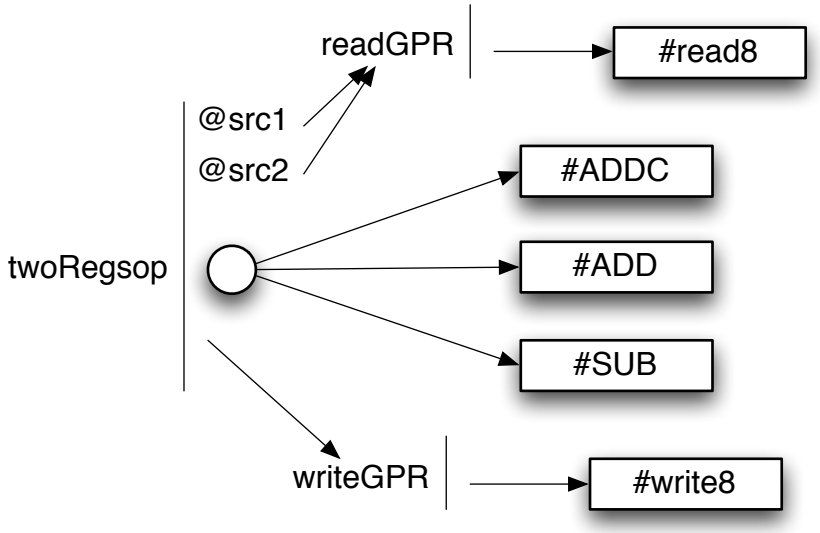



FIGURE 3.3 – Représentation arborescente de l'exemple de la page précédente.

Cet exemple décrit 3 nœuds de type `behavior` : `readGPR`, `writeGPR`, et `twoRegsOp`. Le nœud `twoRegsOp` est la racine de l'arbre, car il n'est appelé dans aucun autre nœud.

L'arbre correspondant à cet exemple est représenté sur la figure 3.3. Chaque nœud est représenté par son nom, ainsi que par une barre verticale. À droite de la barre verticale, les différents « éléments » (structure de sélection, appel d'un autre nœud avec ou sans *post-étiquette*) sont indiqués *séquentiellement*. Le cercle permet de représenter la structure de sélection (`select`). Les *étiquettes* sont représentées dans des rectangles, et les *post-étiquettes* sont représentées dans le nœud appelant.

L'intérêt d'utiliser ici une *post-étiquette* tient dans le fait que le nœud `readGPR` est appelé dans 2 contextes différents : un pour chaque registre source. Comme la *post-étiquette* est ajoutée à chaque *étiquette* du sous-arbre, il y a bien une différenciation : `#read8src1` et `#read8src2`.

Ainsi dans cet exemple, l'instruction *ADD* a comme ensemble d'étiquettes : `#read8src1`, `#read8src2`, `#ADD` et `#write8`. De même, l'instruction *SUB* a comme ensemble d'étiquettes : `#read8src1`, `#read8src2`, `#SUB` et `#write8`. Cet ensemble d'étiquettes constitue la signature de l'instruction correspondante.

#### 3.2.1.3 La vue binaire

La vue binaire permet de décrire comment les instructions sont codées. Cette description permet ensuite de construire un décodeur qui, à partir d'un code exécutable, va construire une représentation interne des instructions de ce programme dans le simulateur.

La description du format des instructions se fait selon une structure arborescente, permettant de mutualiser au maximum les formats communs entre chaque instruction. L'objectif ici est d'extraire à partir du format binaire d'une instruction, à la fois le type de l'instruction et les différents opérandes. Au niveau du simulateur, cette étape constitue le *décodage* de l'instruction.

Dans un premier temps, il est nécessaire de définir dans une section appelée `default` au début de la description d'un processeur, l'information sur la taille des instructions. Par exemple :

```
1 default {
2     instruction := 16  -- taille d'une instruction par defaut
3 }
```

Dans cet exemple, la taille des toutes les instructions du processeur à décrire est sur 2 octets (ou un multiple de 2 octets pour les jeux d'instructions de taille variable). Par exemple, pour le *HCS12*, les instructions ont une taille de 1 à 8 octets. Dans ce cas, il faut préciser dans `default instruction := 8`. La description du format décrit alors la manière de décoder le mot binaire dont la taille est fournie.

La structure générale des nœuds de description des formats est de la forme :

```
1 format <name> <type_options>
2   <formatBody>
3 end format
```

Le champ `<type_options>` est surtout utile dans le cas d'un jeu d'instructions de taille variable (voir plus loin), il permet de préciser quel champ du code binaire de l'instruction est lu dans un nœud donné.

La partie `<formatBody>` est une succession d'éléments de type :

- *étiquette* ;
- appel à un autre nœud de type `format` ;
- structure de sélection, en utilisant le mot clé `select`.
- définition des opérandes.

Par exemple, dans la partie `<formatBody>` de la déclaration suivante, prise de la

description de la vue binaire du co-processeur *XGate* de chez *Freescale*, nous avons tout d'abord une étiquette (`#LSOFF5`), suivi d'une constante (`off5` pour *offset 5 bits*) extraite du code binaire de l'instruction concernée à l'aide de l'opérateur `slice` et formée des 5 bits de poids faibles (4 à 0), suivi d'un appel à deux autres nœuds de type `format` (`loadStoreSize` et `loadStoreType`).

```
format loadStoreWithOffset
  #LSOFF5
  off5 := slice{4..0}
  loadStoreSize
  loadStoreType
end format
```

Le constructeur `select` permet de choisir un sous-nœud parmi plusieurs (ou un ensemble de sous-nœuds parmi plusieurs) en spécifiant aussi un morceau du code binaire utilisant le mot-clé `slice`. Ce morceau est utilisé pour désigner la partie du code de l'instruction qui permettra de distinguer différentes instructions. Dans l'exemple suivant, le mot clé `select` définit plusieurs instructions qui se distinguent par la valeur du bit 12 du code binaire.

```
format loadStoreType
  select slice{12}
    case \b0 is #LOAD
    case \b1 is #STORE
  end select
end format
```

Il est aussi possible d'utiliser le mot clé `or` pour mutualiser plusieurs cas comme dans l'exemple suivant extrait de la description du *HCS12* :

```
select slice {7..0}
    case \x1B or \x1A or \x19 is ...
    ..
```

Dans certain cas, il est intéressant de regrouper plusieurs cas ensembles, le mot clé `others` est alors utilisé comme le montre l'exemple suivant :

```
1  select slice {2..0}
2    case \b110 is ...
3    case \b111 is ...
4    others     is ...
5  end select
```

Dans cet exemple, le troisième choix sera utilisé pour les formats qui ne sont pas du type `11-`. Le mot clé `others` ne peut être utilisé qu'une seule fois, et c'est forcément le dernier cas.

Comme nous l'avons déjà vu, dans le premier exemple, l'extraction des opérandes se fait en utilisant le mot clé `slice`, avec un champ de bits. Ce champ de bits peut être signé ou non signé et a un type implicite qui dépend du nombre de bits qu'il utilise. De plus, quelques opérations de base, comme un décalage, sont possibles quand un champ est extrait. Prenons l'exemple suivant :

```
1 format conditionalBranchInstruction
2   #CondBranch
3   rel10 := signed slice{8..0} << 1 -- 9 bits etendus a 10 par
        le decalage a gauche
4   select slice{12..9}
5     case \b0000 is #BCC
6     case \b0001 is #BCS
7     case \b0010 is #BNE
8     ...
9   end select
10 end format
```

Dans cet exemple, `rel10` est un entier signé de 10 bits (un `s10`) mais n'importe quel nombre de bits peut être utilisé.

Les masques de bits (un nombre binaire préfixé par`\m`) peuvent être utilisés pour indiquer quel morceau `slice` du nombre binaire est non significatif dans la différenciation des instructions. Par exemple, dans :

```
1     ...
2     case \m111- -00- is Indexed_9bits_offset
3     ...
```

Les bits 0 (le signe de l'`offset`), 3 et 4 (le registre utilisé) du morceau `slice` sont utilisés pour décoder le mode d'adressage (comme dénoté par le « - » dans le

masque), et donc non utilisés dans ce cas.

**Cas des jeux d'instructions de taille variable** Dans la description du format, HARMLESS donne la possibilité de décrire des instructions à taille variable d'une manière simple et compréhensible. Dans ce cas, les instructions de taille variable sont décrites en rajoutant dans les nœuds un ou plusieurs mots, relativement aux nœuds précédents. Soit l'exemple suivant, issu de la description du jeu d'instructions du *HCS12*, la taille d'un mot est de 8 bits :

```
1 format Instruction
2   select slice {7..0}
3     case \x18 is inst_18     -- appel a un autre noeud de type
          format
4     ....
5   end select
6 end format
7
8 format inst_18
9   select slice +{7..0}
10    case \m00010111 is #CBA -- 17h
11    ...
12  end select
13 end format
```

Dans cet exemple, le premier nœud (`Instruction`) permet de définir complètement le premier mot de poids fort. Le nœud `inst_18` est appelé si le premier mot est `0x18`. Dans le format `inst_18`, il y a un `+` dans le select, ligne 9 ; ceci indique qu'il est nécessaire de rajouter un mot pour décrire le format binaire de l'instruction. Ainsi, la sémantique de `+{7..0}` est : *1 mot est ajouté, et un* `select` *est fait sur les 8 bits de ce nouveau mot.*

De plus, un nœud dans la vue binaire peut préciser, dans le champ `<type_options>`, quelle partie du code binaire de l'instruction est lue en utilisant aussi le mot clé `slice`. Par exemple, la déclaration suivante, prise de la description du *HCS12* de chez *Freescale*, lit 2 octets. Dans la partie `<formatBody>`, une constante (`offset`) est extraite de ces 2 octets et est la concaténation du bit 4 du premier octet et le deuxième octet.

```
1 format DIT_offset slice {7..0}+{7..0}
2   offset := signed slice{4}{7..0}
3 end format
```

Donc, quand un nœud est allongé en ajoutant un morceau (`slice`), la nouvelle taille est valable pour tous ses nœuds enfants.

Un exemple détaillé sur la description de la vue binaire de quelques instructions d'un jeu d'instructions est donné dans l'annexe A (voir section A.1).

#### 3.2.1.4 La vue syntaxique

La vue syntaxique décrit le format textuel des instructions, en concaténant des chaînes de caractères. En d'autres termes, elle permet d'attribuer une syntaxe textuelle pour chaque signature d'instruction pour permettre, par exemple, le désassemblage.

La vue syntaxique suit le même principe que la vue binaire. C'est un ensemble d'arbres, dans lesquels un nœud décrit une partie de la syntaxe d'une ou plusieurs instructions. Une instruction constitue une branche d'un arbre.

Dans cette vue, chaque nœud suit la syntaxe suivante :

```
1 syntax <name>
2   <syntaxBody>
3 end syntax
```

La partie `<syntaxBody>` est une succession d'éléments de type :

- *étiquette* ;
- déclaration de champ (`field`) ;
- appel à un autre nœud de type `syntax` ;
- structure de sélection, en utilisant le mot clé `select` ;
- chaîne de caractères suivie ou non d'une ou plusieurs variables déclarées.

Comme dans la vue binaire, les noeuds de syntaxe sont associés aux étiquettes qui font partie de la signature des instructions. Par exemple, la syntaxe pour l'instruction `ABA` ($A := A + B$) tirée de la description du processeur `HCS12` (cette signature d'instruction a seulement une étiquette) est :

```
1 syntax ABA
2   #ABA
3   "ABA"  -- chaine de caracteres
4 end syntax
```

La syntaxe de l'instruction `MOVW` est plus complexe et utilise d'autres noeuds de syntaxe qui sont déclarés ailleurs :

```
1 syntax MOVW
2   #MOVW
3   "MOVW"
4   select
5     case imm16 ext
6     case imm16 idx
7     ...
8   end select
9 end syntax
10
11 syntax imm16
12   #IMM16
13   field s16 imm16_value
14   "␣#\d",imm16_value
15 end syntax
```

Cette description signifie que la syntaxe de l'instruction `MOVW` est la concaténation de la chaîne de caractères « `MOVW` », un espace, un `#`, un immédiat de 16 bits et ainsi de suite. Le mot-clé `field` permet de faire référence à un champ qui est extrait de l'instruction dans la vue binaire. Le champ (`field`) doit être typé et est vérifié vis-à-vis de la vue binaire. Prenons comme exemple, un champ de l'instruction non signé nommé `"rs1Index"`, qui donne l'index d'un registre. Dans la vue binaire, il était déclaré de la façon suivante :

```
1 rs1Index := slice{7..5}
```

Dans la vue syntaxique, il sera utilisé sous la forme :

```
1 field u3 rs1Index
```

Ici, nous pouvons remarquer que la taille du `field` est égale à 3 ce qui correspond bien aux 3 bits extraits (bits 7, 6 et 5).

De plus, la vue syntaxique donne la possibilité de construire des structures complexes comme « if...then...else », ce qui est nécessaire pour donner plus de flexibilité dans la syntaxe. Les parties `else` et `elseif` sont facultatives. Cette structure

permet de modifier le comportement dynamiquement (i.e. à la simulation, pour désassembler les instructions, en fonction des valeurs des champs de l'instruction nous choisissons le bon chemin), elle est utilisée notamment pour la description des mnémoniques simplifiés de certaines instructions. Prenons l'exemple ci-dessous.

```
1  syntax orOperation
2    #TriadicInst #OR
3    field u3 rs1Index
4    field u3 rs2Index
5    field u3 rdIndex
6    if rs1Index = rs2Index then
7      "MOV␣R\d,␣R\d", rdIndex, rs1Index
8    else
9      "OR␣R\d,␣R\d,␣R\d", rdIndex, rs1Index, rs2Index
10   end if
11 end syntax
```

Dans cet exemple, selon la condition (les index des deux registres sources sont égaux ou non), une concaténation différente va être faite pendant la phase de désassemblage, par exemple.

Un exemple détaillé sur la description de la vue syntaxique de quelques instructions d'un jeu d'instructions est donné dans l'annexe A (voir section A.2).

#### 3.2.1.5 La vue sémantique

La vue sémantique est la vue la plus complexe des trois, elle permet d'associer un comportement à chaque signature d'instruction. En d'autres termes, elle permet de décrire le comportement des instructions, c'est-à-dire les opérations que les instructions exécutent en accédant aux composants matériels décrits séparément (voir section 3.3) pour pouvoir, comme nous l'avons déjà précisé dans la section 3.1, ajouter des contraintes temporelles sur leur utilisation dans les cas du simulateur précis au cycle près.

Cette vue est formée par un ensemble de nœuds décrivant chacun un comportement ou une partie de comportement d'une ou plusieurs instructions. Un nœud peut faire appel à d'autres nœuds plus élémentaires. Par exemple, un mode d'adressage peut constituer un nœud élémentaire qui sera employé par plusieurs nœuds. Un comportement est rattaché à une instruction via sa signature. Comme les autres vues, la vue comportementale peut comprendre plusieurs arbres, chacun des arbres correspondant à des instructions ayant en commun un ou plusieurs nœuds.

Dans cette vue, un nœud est en définitive semblable à une fonction qui va faire appel à d'autres fonctions (sous nœud ou bien méthodes de composants définies ailleurs dans la description (voir section 3.3)) pour effectuer les opérations nécessaires à l'exécution de l'instruction. Le listing suivant donne le canevas de déclaration d'un nœud de la vue sémantique :

```
behavior <name>(<argumentsList>)
  <behaviorBody>
end behavior
```

Le `<behaviorBody>` est formé d'élements de type :

- *étiquette* ;
- déclaration de variable ;
- déclaration de champ (`field`) ;
- appel à un autre nœud de type `behavior` ;
- structure de sélection, en utilisant le mot clé `select` ;
- blocs `do ... end do` permettant la mise en œuvre d'algorithmes.

La description suivante présente la description d'un ensemble d'instructions, comme par exemple l'instruction `NEG` :

```
behavior mono_operation8(u8 op, out u8 res)
  select
    case #COM  do res := alu.com_8(op);     end do
    case #ASL  do res := alu.sl_8(op);      end do
    case #ASR  do res := alu.asr_8(op);     end do
    case #DEC  do res := alu.add_8(op,0-1); end do
    case #INC  do res := alu.add_8(op,1);   end do
    case #NEG  do res := alu.neg_8(op);     end do
    case #ROL  do res := alu.rol_8(op);     end do
    case #ROR  do res := alu.ror_8(op);     end do
    case #LSR  do res := alu.lsr_8(op);     end do
  end select
end behavior

behavior monadic8_reg_inst
  #INH_AM
  u8 reg;
  u8 res;
  get_reg8(reg)
  mono_operation8(reg,res)
```

```
21   put_reg8(res)
22 end behavior
```

Ici, le nœud `monadic8_reg_inst` déclare 2 variables locales pour stocker le contenu du registre de 8 bits (la source) et le résultat de l'opération. Ensuite, il obtient le contenu du registre de 8 bits du nœud `get_reg8`, exécute les opérations avec le nœud `mono_operation8` (où apparaîssent des appels de méthodes du composant `alu` défini ailleurs dans la description pour faire des additions, des rotations, ...), et stocke le résultat utilisant le nœud `put_reg8`. Dans le nœud `mono_operation8`, une des opérations est choisie. Nous pouvons passer des paramètres d'un nœud à un autre par la référence (utilisant le mot-clé `out`) ou par valeur.

Un exemple plus détaillé sur la description de la vue sémantique de quelques instructions d'un jeu d'instructions est donné dans l'annexe A (voir section A.3).

### 3.2.2 Utilité d'une description divisée en 3 vues

La différence principale entre HARMLESS et les autres *ADL* est la façon dont la description est réalisée. Dans ces *ADL*s, la description est monolithique avec le format binaire (nommé `image` dans *nML*, par exemple), la syntaxe et le comportement (nommés respectivement `syntax` et `action` dans *nML*) partageant le même arbre de description. Dans notre langage, chaque vue a son propre arbre. Suivant cette approche, le concepteur peut choisir la meilleure arborescence pour chaque vue.

Pour illustrer l'avantage de la description dans HARMLESS, nous allons présenter deux exemples pris de la description du processeur *HCS12*. La figure 3.4 montre le sous-arbre de la vue binaire de la famille d'instructions `CALL`. Les formes carrées sont les étiquettes, les barres verticales sont les agrégats et les formes rondes remplacent le mot clé `select`. Comme expliqué dans la section 3.2.1, une branche dans ce sous-arbre correspond à une instruction de la famille `CALL` et est définie par le jeu d'étiquettes dans cette branche. Des masques binaires, utilisés pour décoder les instructions et construire la vue, sont montrés au-dessus des formes carrés. Sur cette figure, tout d'abord une instruction `CALL` est détectée. Après, selon le bit 0 de l'octet du poids faible du code binaire, un mode d'adressage est choisi (immédiat ou indexé). Dans le cas du mode d'adressage immédiat, deux constantes, la première de 16 bits suivie d'une autre de 8 bits, sont extraites du code binaire. Alors que dans le cas du mode d'adressage indexé, un nœud du constructeur `select`, selon les bits de l'octet de poids fort, permet d'extraire l'offset indexé correspondant et ensuite un immédiat de 8 bits est extrait également de cet octet.

La figure 3.5 montre la vue sémantique de la famille des instructions `CALL`

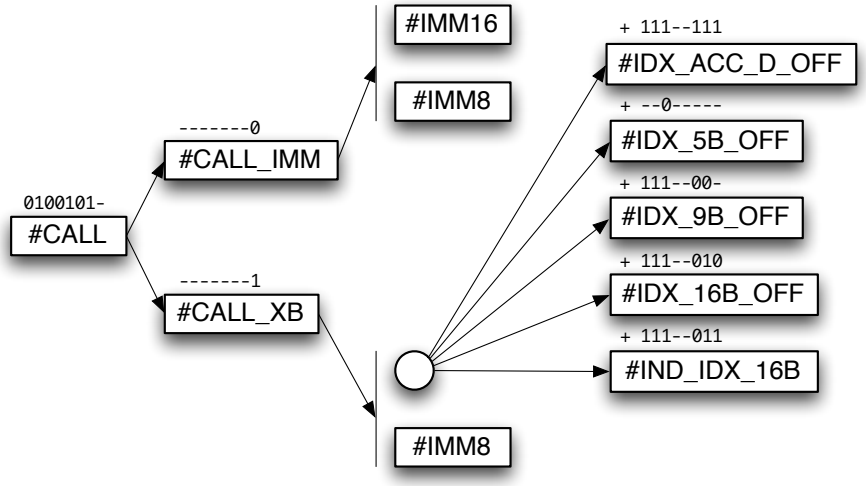



FIGURE 3.4 – Sous-arbre de la description binaire de la famille d'instructions `CALL`.

avec adresse immédiate et `CALL` avec adresse indexée et indirecte. Cette fois la vue a été construite avec le comportement en mémoire et les arbres reflètent les différences sémantiques, puisque le calcul de l'adresse cible est différent dans chaque cas (immédiate, indexée et indirecte).

Comme attendu, il n'y a pas de correspondance directe entre l'arbre de format binaire et l'arbre de comportement. Dans l'arbre de comportement montré sur la figure 3.5, la branche repérée par la lettre `A` correspond aux instructions `CALL` avec le mode d'adressage indexé (l'adresse cible est calculée en additionnant le contenu d'un registre avec une constante) tandis que la branche repérée par la lettre `B` correspond aux instructions `CALL` avec le mode d'adressage indexé indirect (l'adresse cible est lue de la mémoire à une adresse calculée en additionnant le registre `D` et une constante ou donnée comme une valeur immédiate dans l'instruction).

Un autre avantage de cette approche est la capacité de généraliser la description de comportement. Par exemple, la plupart des processeurs n'ont pas un jeu d'instructions orthogonal (toutes les instructions ne fonctionnent pas sur tous les registres) et il est ennuyeux de décrire chaque variante d'accès au registre dans le comportement de chaque instruction. Donc nous pouvons décrire le comportement en supposant que tous les registres sont accessibles par une instruction donnée. Quand la description est compilée, HARMLESS enlève tous les comportements qui n'ont pas de format correspondant. Nous le montrons dans l'exemple ci-dessous.

La figure 3.6 montre l'arbre de format de l'instruction de transfert `TFR`. Cette instruction transfère les données d'un registre source à un registre destination . Le registre `TMP2` peut seulement être une destination tandis que le registre `TMP3` peut

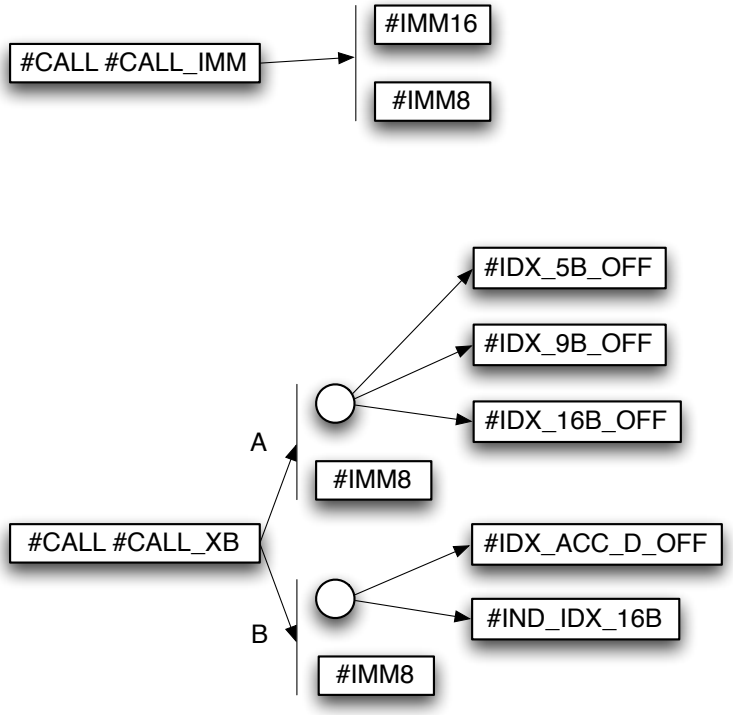



FIGURE 3.5 – Sous-arbres de la vue sémantique de la famille des instructions `CALL` avec adresse immédiate et des instructions `CALL` avec adresse indexée ou indirecte.

seulement être une source.

La figure 3.7 montre l'arbre de comportement de l'instruction `TFR` (`TFR 16-16` où les registres source et destination sont des registres 16 bits, `TFR 8-8` où les registres source et destination sont les registres 8 bits $A$,$B$ et $C$, `TFR 16-8` où le registre source est 16 bits alors que le registre destination 8 bits et `TFR 8-16` où le registre source est 8 bits et le registre destination 16 bits).

Le comportement décrit des instructions qui n'existent pas (toutes les instructions `TFR` avec `TMP2` comme registre source ainsi que les instructions `TFR` avec `TMP3` comme registre destination) mais puisque le format pour de telles instructions n'existe pas, le simulateur produit par HARMLESS n'inclut pas ces instructions.

Ces caractéristiques permettent de simplifier la description. Comme résultat, la description du *HCS12* utilisant *nML* nécessite 7400 lignes alors que celles utilisant HARMLESS nécessite 2700 lignes.

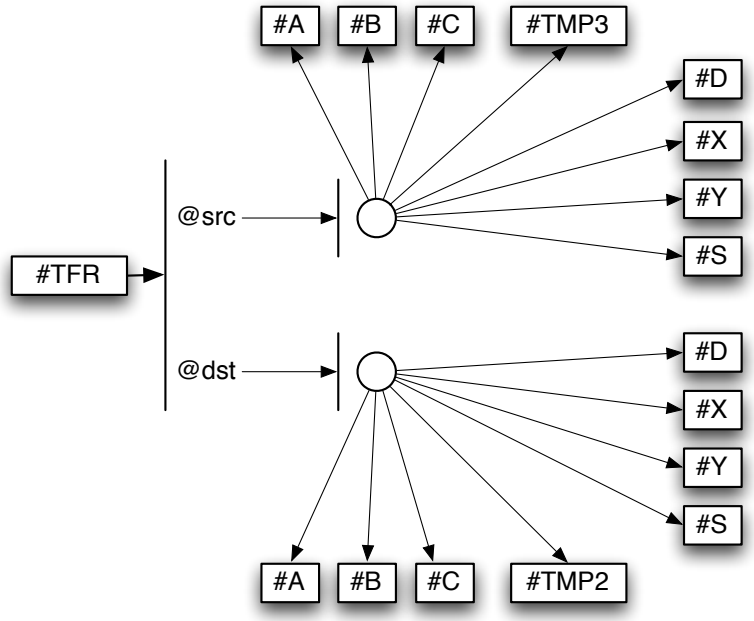



FIGURE 3.6 – Arbre de format de l'instruction TFR.

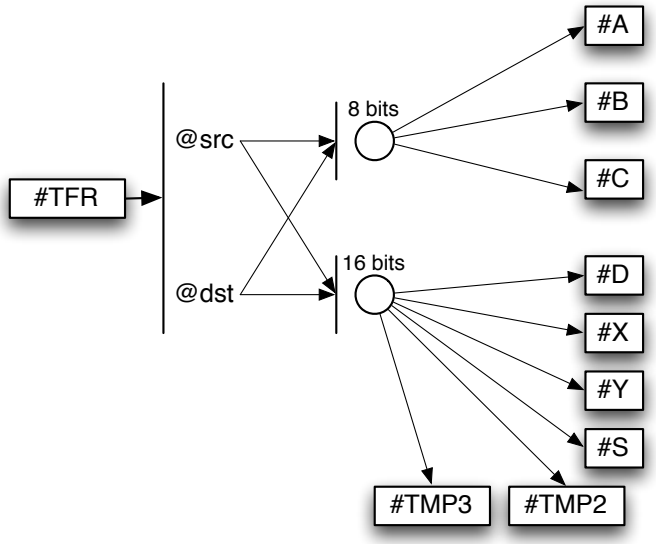



FIGURE 3.7 – Arbre de comportement de l'instruction TFR.

## 3.3 Description fonctionnelle des composants matériels

Dans cette section, nous allons exposer la façon dont HARMLESS permet la description des différents composants matériels du point de vue fonctionnel (la vue temporelle sera présentée dans la partie II).

Un composant est l'unité élémentaire de structure de l'architecture d'un processeur comme les registres, la mémoire ou l'unité arithmétique et logique, abrégée *UAL* (ou bien *ALU* pour *Arithmetic Logic Unit*). La description d'un composant est encapsulée et contient :

- des variables membres ;
- des méthodes associées.

Au niveau d'un *behavior* (voir section 3.2.1.5), les différentes méthodes d'un composant peuvent être appelées, à l'intérieur d'un bloc « `do .. end do` » de la manière suivante : `<componentName>.<methodName>`.

Soit par exemple un composant `Fetcher` chargé de la gestion du compteur programme :

```
1  component Fetcher {
2    program counter u16 pc; -- genere les methodes 'get' et 'set'
3
4    void reset() {
5      pc := 0;
6    }
7
8    void branch(s16 offset) {
9      pc := (u16)((s16)(pc) + offset);
10   }
11 }
```

Ce composant utilise une variable membre, qui est un registre (ici un registre spécifique : le compteur programme). Il y a aussi 2 méthodes qui y sont associées `reset` et `branch`, dont la syntaxe ressemble beaucoup à celle du C. Nous pouvons accéder à ce composant dans la vue comportementale de la manière suivante : `Fetcher.branch(10)`.

### La description de la mémoire

La description des éléments de mémorisation est nécessaire pour la génération d'un simulateur de jeu d'instructions, mais aussi d'un simulateur précis au cycle.

Dans cette dernière approche, il sera de plus nécessaire de rajouter des informations relatives au comportement temporel de la mémoire.

Dans HARMLESS, la description de la mémoire est encapsulée dans un composant. Ceci présente un double avantage :

- il est possible de rajouter des méthodes dans le composant, afin de simplifier l'accès à la mémoire (rajout de méthodes `push/pop` pour modéliser une pile, modélisation de mémoire paginée, ...) ;
- il est aussi possible de permettre l'accès à plusieurs zones de mémoire de manière transparente. Par exemple, une zone de mémoire RAM (largeur 16 bits) et une EEPROM (largeur 8 bits) dans le même espace d'adressage.

Une zone de mémoire est déclarée en utilisant le mot clé `memory`. Certains paramètres permettent de définir la zone mémoire comme (voir l'exemple suivant) :

- la largeur du bus ;
- la plage d'adresses ;
- le décalage (voir exemple page suivante) ;
- le type de la mémoire (`RAM` dans le cas où la zone peut être accédée en lecture et écriture, `ROM` dans le cas où la zone peut être accédée seulement en lecture ou `register` qui a pour le moment la même signification que le type `RAM`).

```
1 component mem {
2   program memory internalRam {
3     width   := 16 -- permet d'obtenir maximum 16 bits / Acces
4     address := \x0..\xFFFF
5     type    := RAM
6   }
```

Dans cet exemple, une zone mémoire de type `RAM` est définie sur la zone d'adresse qui commence à `\x0` et va jusqu'à `\xFFFF` inclus. L'accès à cette zone sera possible sur 16 bits mais aussi sur les tailles de mots inférieurs, ici 8 bits.

La mémoire peut être accédée en écriture (sauf la mémoire de type `ROM`) et en lecture en utilisant des méthodes implicites. En se basant sur l'exemple ci-dessus decrivant le composant `mem`, 4 méthodes prédéfinies peuvent être utilisées qui sont :

- `u16 mem.read16(u32 address)` renvoie la valeur d'un élément dans le composant, sur 16 bits. Suivant l'adresse, une valeur de la zone mémoire `mem` sera renvoyée. Si l'adresse est invalide (ne correspond pas à une zone mémoire), la valeur 0 est renvoyée ;
- `u8 mem.read8(u32 address)` renvoie la valeur d'un élément dans le composant, sur 8 bits, de la même manière que la méthode précédente ;
- `void mem.write16(u32 address, u16 data)` écrit la valeur `data` à l'adresse `address` de l'élément mémoire `mem`. En cas d'adresse invalide, une

erreur sera générée ;

– `void mem.write8(u32 address, u8 data)` idem sur 8 bits.

Le mot clé `program` est utilisé pour déterminer la mémoire qui peut accepter un code programme au démarrage. Il n'est pas possible d'avoir 2 zones mémoires avec une plage d'adresses qui se superpose en utilisant le mot clé `program` (il ne doit pas y avoir de recouvrement). En effet, lors du chargement du programme exécutable dans la zone mémoire du simulateur, il ne doit pas avoir d'ambiguïté sur la zone mémoire cible. Par exemple, dans une architecture de type *Harvard*, qui sépare la mémoire de données de la mémoire programme, une plage d'adresses des deux mémoires peuvent se superposer et c'est pour cette raison qu'il faut les distinguer en précisant laquelle des deux mémoires va stocker le programme à simuler.

De plus, il est possible de faire un décalage des adresses, à l'aide du mot clé `stride`. Il permet de proposer une utilisation intuitive des accès aux registres, ou plus généralement à la mémoire, quand nous souhaitons y accéder non pas octet par octet, mais au niveau du mot (typiquement 16 ou 32 bits). Le stride doit être une puissance de 2. L'adresse réelle est obtenue en ajoutant à l'adresse de départ le produit du déplacement à effectuer (par exemple, numéro du registre à accéder) par la valeur du paramètre `stride` :

*adresse_réelle = adresse_départ + déplacement * stride*

Soit par exemple une zone de 16 registres généraux (adressés en mémoire) sur un processeur 16 bits définis de la manière suivante :

```
1  memory GPR {
2    width   := 16    -- permet d'obtenir 16 bits / acces
3    address := 0..31 -- 32 octets : 16 registres (chacun de
         16 bits)
4    stride  := 2
5    type    := register
6  }
```

Ainsi dans cet exemple, l'accès au registre `5` permettra une lecture dans la mémoire à l'adresse `10 = 0 + 5 * 2` (au lieu de 5) qui contient réellement le contenu du registre GPR 5.

### Sous blocs mémoire

À l'intérieur d'une zone mémoire, il est possible de définir un sous bloc qui va avoir des caractéristiques différentes. Par défaut, un sous-bloc *hérite* des paramètres du bloc dans lequel il est inséré.

D'autre part, il est possible de mapper le sous-bloc à l'intérieur du bloc prin-

cipal. Par exemple :

```
1  memory ram {
2    width   := 16  -- permet d'obtenir 16 bits / acces
3    address := \x0..\x10FF
4    type    := RAM
5
6    sfr { -- 'Special Function Register' peut etre accede par
         les instructions IN/OUT
7      address := 0..\x3F
8      type    := register
9    } maps to \x20
10 }
```

Cet exemple tiré de la description de l'AVR met en évidence :

- la plage mémoire de `0` à `0x10FF` est définie comme de la RAM ;
- la plage mémoire de `0x20` à `0x20+0x3F` est *redéfinie* comme étant de type `register` (en l'absence du `maps to <expression>`, le sous-bloc est mappé à l'adresse 0) ;
- un accès à l'adresse `0x30` de la zone *ram* correspond à la même place qu'un accès à l'adresse `0x10` (à cause de l'offset de `0x20`) d'un *sfr*.

Ce dernier point est le plus important, car lors de la description du jeu d'instruction, il ne sera pas nécessaire de faire de décalage dans la *ram* lorsqu'on fera un accès à un registre de type *sfr*.

### Les registres

Les registres sont utilisés à de nombreux endroits dans la description, que ce soit dans les composants, la mémoire, et même la vue comportementale de la description.

C'est pourquoi à la première vue, nous pouvons croire que les registres sont dans une certaine mesure une entorse à l'encapsulation qui est présentée dans les composants. Or, c'est uniquement une simplification syntaxique, pour éviter de surcharger la description ; les accès aux registres sont en fait réalisés à travers des méthodes d'accès (`get/set`) qui sont générées de manière implicite.

**Définition dans un composant** Dans un composant, il est déclaré en utilisant le mot clé `register` :

```
1 register u8 SP
```

Cet exemple défini un registre nommé SP (de 8 bits) qui est accessible dans tous les composants, en utilisant directement son nom, dans une zone d'implémentation :

```
1  SP := SP+1
```

Un registre peut être défini en nommant des champs de bits comme le montre l'exemple ci-dessous.

```
1  component alu {
2    register u8 CCR {
3      C := slice{0} -- carry flag
4      V := slice{1} -- overflow flag
5      Z := slice{2} -- zero flag
6      N := slice{3} -- neg flag
7      I := slice{4} -- maskable interrupt
8      H := slice{5} -- half carry status bit
9      X := slice{6} -- non-maskable interrupt
10     S := slice{7} -- STOP instruction
11   }
12
13   u8 neg_8(u8 op) {
14     u8 res;
15     res := 0 - op;
16     CCR.N := res{7};
17     CCR.Z := res = 0;
18     CCR.V := op = \x80;
19     CCR.C := op != 0;
20     return res;
21   }
22   ...
23 }
```

Dans cet exemple, une partie du composant `alu` et une de ses opérations pour le modèle *HCS12* sont décrites. Le composant `alu` encapsule la description du registre d'état (ou bien `CCR` pour Condition Code Register) en nommant des champs de bits que nous pourrons mettre aussi dans un « component » à part. L'accès à un champ de bits se fait alors de la manière suivante :

```
1  CCR.V := 1;
```

**Définition dans un bloc mémoire** Un registre défini dans une zone mémoire est mappé en mémoire, il est donc nécessaire de préciser à quelle adresse il est mappé :

```
1  component sram {
2    memory ram {
3      width   := 16  -- permet d'obtenir 16 bits / acces
4      address := \x0..\x10FF
5      type    := RAM
6
7      register u8  SPH  maps to \x5e  -- pointeur de pile (octet
           de poids fort)
8      register u8  SPL  maps to \x5d  -- pointeur de pile (octet
           de poids faible)
9      register u16 SP   maps to \x5d  -- pointeur de pile (16
           bits)
10     ...
11   }
12 }
```

Si la taille du registre n'est pas spécifiée, c'est la largeur de bus qui est utilisée (en non signé).

Ainsi, les 2 lignes suivantes sont identiques :

```
1  SP := SP+1
2  sram.ram.write16(\x5d, sram.ram.read16(\x5d)+1)
```

## 3.4 Conclusion

Ce chapitre a présenté les principes fondamentaux permettant la description du jeu d'instructions d'un processeur en utilisant HARMLESS.

Dans un premier temps, la description du jeu d'instructions a été présentée en montrant comment HARMLESS permet une description incrémentale en séparant cette description en trois vues : binaire, syntaxique et sémantique et dans un second

temps la description fonctionnelle des composants matériels a été abordée. Ceci permet déjà de générer un simulateur de jeu d'instructions.

Dans le chapitre suivant, nous allons parler de la génération du simulateur *ISS*, en expliquant comment sont modélisées les instructions. De plus, quelques résultats seront donnés et analysés.

# Chapitre 4

# Génération du simulateur fonctionnel

## 4.1 Introduction

Dans le chapitre précédent, nous avons expliqué comment décrire un processeur en utilisant HARMLESS pour pouvoir engendrer automatiquement un simulateur *ISS*. Ce chapitre va donner quelques détails sur la génération automatique d'un tel simulateur, comme l'implémentation et le décodage des instructions.

Dans un premier temps, le modèle utilisé pour les instructions sera expliqué ainsi que la génération du décodeur. Dans un second temps, une amélioration basée sur l'utilisation d'un cache logiciel d'instructions sera présentée, pour accélérer la simulation. Enfin, quelques résultats sur le processus de génération du simulateur à partir de plusieurs descriptions de processeurs seront analysés.

## 4.2 Implémentation des instructions

Dans la description HARMLESS, une instruction correspond à un chemin dans un arbre (voir section 3.2.1.1). Cette approche permet de mutualiser les parties communes entre différentes instructions. Cependant, au moment de la génération du simulateur la structure arborescente doit être mise à plat et il y a beaucoup de code généré qui est dupliqué. En effet, même si nous utilisons la mutualisation de parties communes dans les descriptions, il y a une « mise à plat » (et donc duplication) du code correspondant lors de la génération du simulateur. Ceci permet de supprimer le temps nécessaire à l'appel des fonctions dans le simulateur généré dans lequel chaque instruction sera modélisée comme une classe *C++*.

La classe *C++* qui représente une instruction offre 3 méthodes principales (un exemple détaillé est présenté dans l'annexe A.4) :

– *Le constructeur* Il est associé à l'opération de décodage. Son but est d'identifier les différents champs du code binaire des instructions (index des registres, valeur immédiate, adresse). Considérons, par exemple, une instruction d'addition d'une architecture *RISC* : `ADD R1, R2, R3`. Supposons que le code binaire de cette instruction sur 16 bits soit le suivant :

| | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD RD, RS1, RS2 | 0 | 0 | 0 | 1 | 1 | RD | | | RS1 | | | RS2 | | | 1 | 0 |

À partir de ce code binaire, le constructeur extrait : l'index du registre destination `(1)` à partir des bits (`10`, `9` et `8`), et les indices des deux registres sources `(2)` et `(3)` à partir des bits (`7`, `6` et `5`) et (`4`, `3` et `2`) respectivement. Une propriété importante tient au fait que cette opération est totalement indépendante du contexte du simulateur. Cette fonction est directement attachée à la description binaire dans HARMLESS ;

– *La fonction d'exécution* Son rôle est de simuler l'exécution entière de l'instruction. Dans HARMLESS, elle est associée directement à la description sémantique. En utilisant l'exemple précédent basé sur l'instruction d'addition, cette fonction va lire les valeurs des deux registres sources (`R1 et R2`), additionner les deux valeurs, mettre à jour le registre d'état et enfin écrire le résultat dans le registre destination (`R3`). À noter que si pour une instruction donnée il n'y a pas de description sémantique, une description par défaut sera utilisée générant un message à l'utilisateur pendant la simulation [1] ;

– *La fonction mnémonique* Elle renvoie une chaîne de caractères comportant le mnémonique de l'instruction. Elle est associée à la description syntaxique dans HARMLESS et est utilisée pour le désassemblage. De la même manière que le constructeur, cette fonction ne modifie pas le contexte du simulateur. À noter, si pour une instruction donnée il n'y a pas de description syntaxique, une description par défaut, qui retourne le nom interne de l'instruction, sera utilisée.

L'exécution d'une instruction est basée initialement sur l'approche interprétée (voir section 2.2.2.4), exécutant chaque instruction l'une à la suite de l'autre. Le processus d'exécution est donné sur la figure 4.1. D'abord, la phase de décodage va décoder le code binaire pointé par le pointeur d'instruction (expliqué en détail dans la section 4.2.1). L'instance d'instruction est créée (nécessitant une allocation mémoire). Ensuite, l'instruction est exécutée et enfin, l'instance d'instruction est supprimée (nécessitant une désallocation mémoire). Cette phase doit être traitée pour chaque instruction dans le programme et l'allocation/désallocation mémoire pénalise particulièrement le temps de calcul. Pour améliorer les performances, une

1. C'est particulièrement utile quand la description est faite de manière incrémentale car cela permet de générer un simulateur avec un jeu d'instructions incomplet.

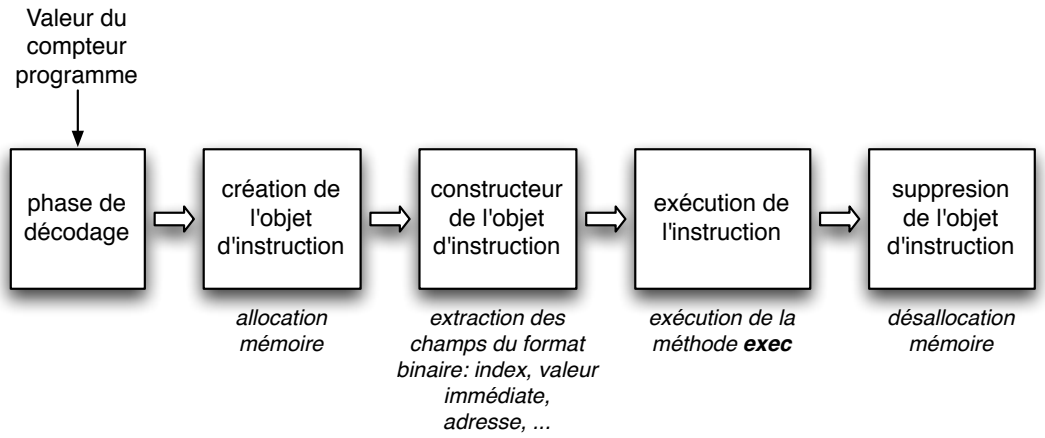



FIGURE 4.1 – Exécution d'une instruction.

approche plus efficace, en utilisant un cache logiciel d'instructions est expliquée dans la section 4.2.2 .

### 4.2.1 Le décodage (Génération du décodeur)

Basé sur la partie de description de format (section 3.2.1.3), la génération du décodeur est une partie importante de la génération du simulateur. Puisqu'une instruction est représentée comme une branche dans un arbre, la première opération doit aplatir l'arbre et extraire, pour chaque instruction, tous les formats utilisés dans la description. À chaque format, un couple masque/valeur permet d'identifier les bits significatifs et les bits qui ne sont pas représentatifs.

Considérons, par exemple, l'exemple ci-dessous. Pour toutes les instructions, qui contiennent le nœud d'instruction `TBL_inst`, le masque/valeur sera `0xF/0xD`. C'est une condition qui signifie que seulement les quatre bits de poids faible sont significatifs (le masque est `0xF` correspondant à {3..0}) et la valeur donne l'état de ces 4 bits. Le code final de l'instruction est représenté par la conjonction de toutes les conditions (masque/valeur) liées à chaque partie de format.

```
format inst_183x
  select slice {3..0}
    case \xA is #REV
    case \xB is #REVW
    case \xC is #WAV
    case \xD is TBL_inst
    ...
  end select
end format
```

Pour faciliter les traitements, la représentation interne des conditions (masque/valeur) est codée en utilisant des Diagrammes de Décision Binaires (BDD) [2]. Cela permet de vérifier d'une manière très simple l'orthogonalité du jeu d'instructions : si deux instructions peuvent avoir le même code binaire, alors la conjonction de leur BDD respectifs n'est pas un BDD vide. Il suffit de faire la conjonction des BDDs de toutes les combinaisons d'instructions deux à deux pour vérifier l'orthogonalité du jeu d'instructions.

Avec l'utilisation interne des BDDs, nous obtenons des conditions simples, indépendamment de la description sous-jacente. La condition est appliquée sur le code binaire d'instruction pointé par le compteur de programme. Voici un exemple avec 2 conditions qui peuvent correspondre à une instruction, mais la plupart des instructions sont décodées en utilisant seulement une condition :

```
if(((code & mask1) == value1) ||
    (code & mask2) == value2)) {
        //instruction found
}
```

Le décodeur peut être directement généré en utilisant une liste de conditions comme celle-ci, pour chaque instruction. La taille des masques et des valeurs est indépendante de la taille du code d'instruction et est définie pour réduire au minimum le traitement sur la machine hôte. Pour le `HCS12` par exemple, la taille de l'instruction est comprise entre 1 et 8 octets. Pour un simulateur s'exécutant sur une machine 32 bits, si la taille de l'instruction est inférieure à 5 octets, une seule comparaison est nécessaire. Sinon, une deuxième comparaison, également sur 32 bits, serait nécessaire pour décoder le reste du code d'instruction.

Ainsi, un décodeur peut être généré en une seule fonction qui intègre toutes ces comparaisons comme le montre la figure 4.2 (à gauche de la flèche). Si la comparaison permettant de déterminer l'instruction en cours de décodage est présente à la fin de cette grande fonction, de nombreux tests inutiles seront nécessaires pour

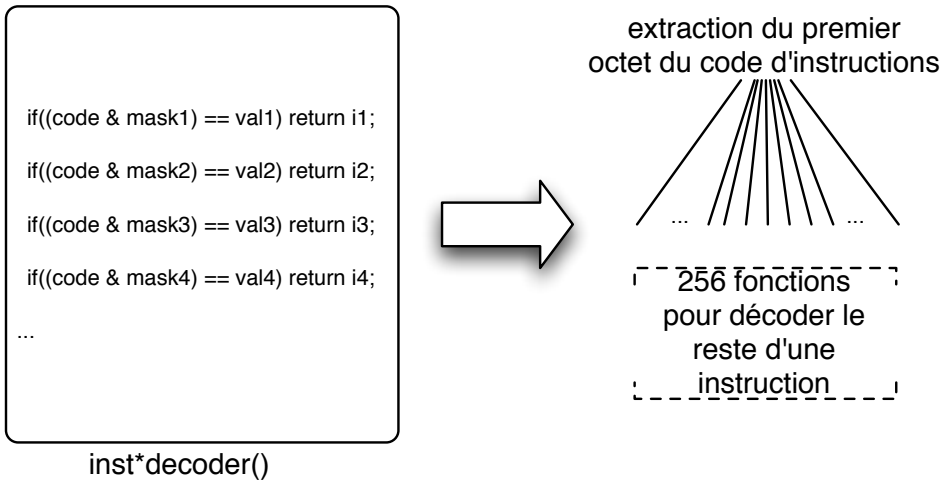



FIGURE 4.2 – Le décodage d'instructions.

décoder l'instruction. Afin de minimiser le nombre de comparaisons, nous avons fait l'hypothèse que dans la plupart des jeux d'instructions, une partie du code binaire utilisée pour le décodeur (c'est le code opération) est définie dans l'octet du poids fort du code binaire. Puis, comme le montre la figure 4.2 (à droite de la flèche), selon le premier octet du code d'instruction, $2^8 = 256$ fonctions sont générées afin de décoder le reste de l'instruction. Cette hypothèse permet de minimiser considérablement le nombre de comparaisons nécessaires dans les exemples étudiés (*AVR* d'*Atmel*, *HCS12* de *Freescale*, *XGate* et *PowerPC*). Le nombre de bits utilisés pour générer des fonctions sous-décodeur a été fixé à 8 par défaut, mais peut être modifié. Par exemple, avec le *pic10 Microchip* (la taille de l'instruction est de 12 bits), la valeur peut être augmentée à 12, pour obtenir $2^{12} = 4096$ fonctions sous-décodeur.

### 4.2.2 Optimisation de la vitesse de simulation fonctionnelle

Le processus d'exécution basé sur l'approche classique interprétée, présenté dans la figure 4.1, comporte deux inconvénients majeurs de nature structurelle :

- Lors de la simulation d'une instruction dans une boucle, l'instruction sera décodée à plusieurs reprises ;
- L'allocation/désallocation de la mémoire représente la plus grande partie du temps de calcul, lors de la création et la suppression de l'objet de l'instruction en langage `C++`.

Or, du fait de la présence des boucles, l'exécution d'un programme présente une forte localité temporelle. Afin d'améliorer l'approche précédente, nous avons donc ajouté un cache logiciel d'instructions au cours de la phase de décodage. Cette approche est une version simplifiée de la simulation compilée du jeu d'instructions (*Instruction Set Compiled Simulation*) dans [53]. Le cache logiciel d'instructions est interne, il n'a que peu d'effets secondaires : la mémoire contenant le programme à simuler ne doit pas être modifiée lors de la simulation car la modification ne serait pas prise en compte ; nous pourrions la prendre en compte en invalidant les instructions correspondantes dans le cache, mais ce n'est pas fait à l'heure actuelle. Ce type de situation ne se rencontre pas pour les programmes visés (temps réel embarqué), et cette technique conserve tous les avantages liés aux approches de simulation interprétée (voir section 2.2.2.4).

Le principe de la simulation avec un cache logiciel d'instructions est décrit sur la figure 4.3. Quand une instruction est décodée la première fois, le cache logiciel d'instructions retourne un échec (*miss*) et un objet *C++* de l'instruction est créé comme dans l'approche précédente. Les nouvelles instructions sont stockées dans le cache (la taille du cache est fixe et est égal à 32768). La prochaine fois que l'instruction doit être exécutée, l'objet de l'instruction est dans le cache (la cache retourne un succès *hit*). De cette manière, le temps de simulation est réduit car les phases de décodage, d'allocation de l'objet et de construction sont éliminées.

Au niveau de la description, ce qui est décrit dans la partie format correspond au décodage et à la fonction qui construit l'instance de l'objet. En utilisant un cache logiciel d'instructions, cette opération n'est effectuée que la première fois qu'une instruction est rencontrée (sous réserve qu'elle ne soit pas exclue du cache).

Dans notre simulateur, le cache logiciel d'instructions utilisé est une mémoire cache à correspondance directe (*direct mapped cache*). Ceci est le plus simple à mettre en œuvre car d'une part il n'y a pas de politique de remplacement et d'autre part, il offre des résultats intéressants.

## 4.3 Expérimentation et analyse des résultats

Dans cette section, nous allons présenter quelques résultats sur le processus de génération des simulateurs de plusieurs processeurs :

- Le *HCS12* qui est un microcontrôleur *CISC* de *Freescale* avec un jeu d'instructions de taille variable (de 1 jusqu'à 8 octets) ;
- Le *PowerPC* qui est un processeur basé sur une architecture *RISC* avec une taille d'instructions fixe de 32 bits ;
- La *XGate* qui est un co-processeur de chez *Freescale* qui est intégré dans le microcontrôleur *HCS12X*. Elle est basée sur une architecture *RISC*, avec une taille d'instruction fixe de 16 bits ;

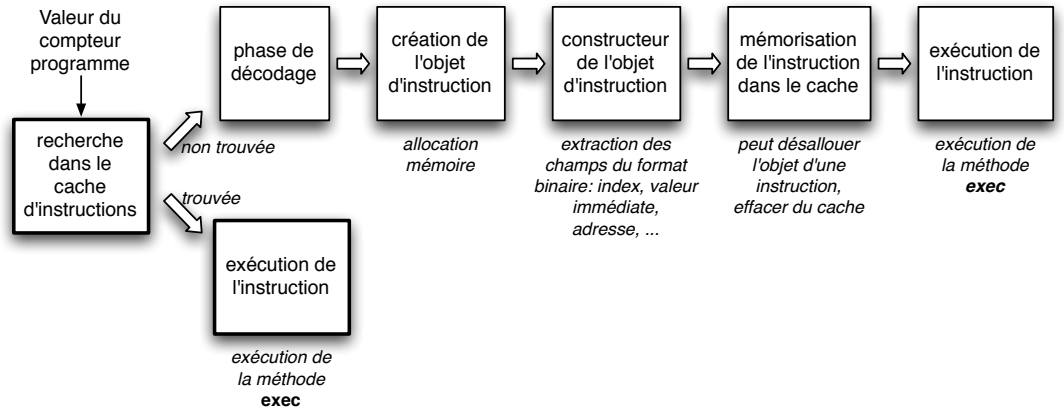



FIGURE 4.3 – Exécution d'une instruction en utilisant un cache logiciel d'instructions.

– L'*AVR* d'*Atmel* qui est un processeur basé sur une architecture *RISC* avec une taille d'instruction fixe de 16 bits (même si quelques instructions ont une taille de 32 bits).

Les expérimentations sont basées sur un exemple simple permettant le calcul de la suite de Fibonacci (voir tableau 4.1). Cela donne un aperçu sur les performances du simulateur généré automatiquement à partir de la description de chaque processeur en utilisant HARMLESS. À noter que les simulateurs *ISS* générés à partir de la description des deux processeurs *AVR* et *PowerPC* dans HARMLESS sont les plus matures, ils ont notamment été validé sur la simulation du *RTOS Trampoline*[2].

Les processeurs basés sur une architecture *RISC* ont moins d'instructions que celui basé sur une architecture *CISC* (plus de 5500 instructions). Ce n'est pas seulement dû au nombre important de modes d'adressage, mais aussi à cause de l'architecture du *HCS12*. En effet, Il y a peu de registres dans l'architecture de *HCS12*, par exemple, l'instruction de rotation à gauche est assurée par deux instructions `ROLA` et `ROLB`, selon le registre envisagé (`A` ou `B`). Dans la description, nous pouvons soit décrire une instruction `ROLx` avec un paramètre champ (`field`) qui permet de choisir entre les deux registres (`A` ou `B`), ou bien décrire les deux instructions différentes. Nous avons choisi la deuxième solution pour profiter des avantages de la mémoire cache d'instructions interne : augmentation de la complexité de décodage et simplification de la description du comportement (et donc augmentation de la vitesse de simulation).

2. *RTOS* compatible *OSEK OS* développé à l'*IRCCyN* (`http://trampoline.rts-software.org/`).

| | **HCS12** | **PowerPC** | **XGate** | **AVR** |
|---|---|---|---|---|
| Longueur de la description (lignes) | 2925 | 3208 | 1139 | 1408 |
| Instructions générées (nombre) | 5590 | 332 | 88 | 90 |
| Temps pour engendrer les sources du simulateur à partir de la description en HARMLESS (s) | 30.39s | 4.3s | 0.38s | 0.42s |
| La taille des sources du simulateur (lignes en C++) | $\sim$ 418343 | $\sim$ 40970 | $\sim$ 11980 | $\sim$ 11622 |
| Temps pour compiler le simulateur (s) | 545.16s | 32.38s | 12.30s | 11.92s |
| Temps pour exécuter 100 millions d'instructions de l'exemple de base (s) | 3.17s | 3.16s | 3.74s | 3.83s |
| Temps pour exécuter 100 millions d'instructions de l'exemple de base sans cache logiciel d'instructions (s) | 18.58s | 14.58s | 20.14s | 14.21s |

TABLE 4.1 – Ce tableau présente quelques résultats sur la génération du simulateur. Le calcul de temps a été fait sur un *Intel core 2 Duo* à *2 GHZ*.

Comme le *HCS12* a beaucoup d'instructions, ceci conduit à augmenter le temps nécessaire pour engendrer le simulateur et compiler les fichiers générés en *C++*, mais cela n'est fait qu'une seule fois. Nous pouvons remarquer que si la vérification d'orthogonalité du jeu d'instructions est désactivée (elle n'est pertinente que lors de la description de la partie format), le temps pour générer les sources du simulateur est réduit à 18.68$s$. Un simulateur *ISS* est construit en moins de 10 minutes pour une architecture *CISC* et en moins de 40 secondes pour une architecture *RISC* partant de la description et aboutissant au simulateur de l'application binaire.

Le processus de génération du simulateur est exécuté une seule fois pour chaque modèle. Donc, le point le plus important à comparer se porte sur la vitesse de simulation qui se réfère aux deux dernières lignes du tableau 4.1. Nous pouvons remarquer que les temps de calcul nécessaires pour les quatre modèles sont dans le même ordre de grandeur (moins de 4 s avec un cache d'instructions et de 20 s sans cache). Ce dernier résultat montre l'importance d'un cache logiciel d'instructions qui permet d'éviter le décodage multiple d'une même instruction : la suppression du cache d'instructions interne engendre une perte de performance de 80%.

## 4.4 Conclusion

Dans ce chapitre, nous avons présenté les étapes de génération automatique du simulateur fonctionnel *ISS*. Le simulateur est engendré dans le langage de programmation objet *C++*.

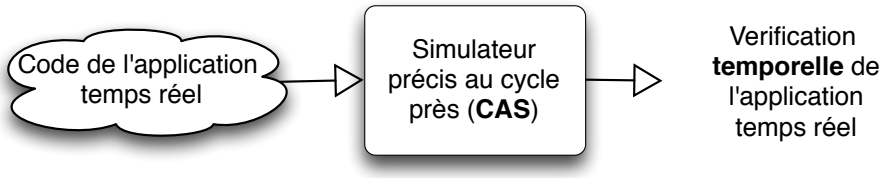
Trois points importants étaient abordés tout au long de ce chapitre : la modélisation d'instructions, la génération du décodeur et l'importance d'un cache logiciel d'instructions. Chaque instruction est modélisée en une classe *C++* offrant 3 méthodes principales : le constructeur (associé à l'opération du décodage), la fonction d'exécution (simule l'exécution entière des instructions) et la fonction mnémonique (utilisée pour le désassemblage). Le décodeur est généré à partir de la description du code binaire (format) de jeu d'instructions en se basant sur l'utilisations des BDDs. L'utilisation d'un cache logiciel d'instructions rend l'exécution d'un programme plus rapide en évitant le décodage d'une instruction plusieurs fois. Enfin, les résultats obtenus montrent les bonnes performances des simulateurs générés automatiquement à partir de la description de plusieurs processeurs (*PowerPC*, *HCS12*, *XGate* et *AVR*).

# Deuxième partie

# Simulateur précis au cycle près ($CAS$)

La simulation du matériel est un élément important dans la conception des systèmes embarqués notamment temps réel. Par exemple, elle permet de fournir un moyen d'exécuter un logiciel quand le matériel final n'est pas encore disponible, ce qui peut être utilisé pour calculer le pire temps d'exécution ( *WCET* ) d'un code, et/ou analyser les comportements temporels d'une application.

Pour pouvoir satisfaire ces besoins, une description à la fois fonctionnelle et temporelle du processeur est nécessaire. Cette description nous permettra de générer un simulateur connu sous le nom d'un simulateur précis au cycle près ( *CAS* ) ; ce sera le thème autour duquel va se développer cette partie. Un tel simulateur, comme le montre la figure ci-dessous, accepte en entrée le code d'une application temps réel, et l'exécute. En sortie, des caractéristiques temporelles pourront être obtenues, amenant ainsi à la vérification temporelle de l'application temps réel.




Dans cette partie, nous allons expliquer comment, en utilisant notre langage de description d'architecture matérielle HARMLESS, un simulateur *CAS* peut être généré automatiquement. Pour cela et afin de l'obtenir, il faut ajouter à la description présentée dans la partie I, la description de la micro-architecture.

Cette partie est formée de trois chapitres. Dans le premier, nous allons présenter le modèle interne d'un pipeline simple (un automate est utilisé pour modéliser le comportement du pipeline). Dans le deuxième chapitre, la technique utilisée pour que notre langage HARMLESS permette la description de la micro-architecture (le pipeline et les concurrences d'accès aux composants matériels) sera expliquée. Dans le troisième chapitre, les différentes étapes de la génération du simulateur précis au cycle près seront présentées. À noter que dans les chapitres il y aura des exemples tirés de la description de plusieurs processeurs : le *XGate* (avec un pipeline fictif) qui est le co-processeur du *HCS12* et le *PowerPC 5516*.

# Chapitre 5

# Modélisation d'un pipeline simple

## 5.1 Introduction

La description du jeu d'instructions qui a été traitée dans la partie précédente permet d'engendrer un simulateur fonctionnel, c'est-à-dire que le simulateur généré sera capable de décoder, désassembler et exécuter un programme à partir de son code objet. Cependant, aucune information temporelle n'est disponible. La description de la micro-architecture du processeur, ajoutée au jeu d'instructions préalablement décrit, a pour objectif d'engendrer un *simulateur précis temporellement* connu sous le nom de *CAS* (*Cycle Accurate Simulator*).

L'une des caractéristiques architecturales coûteuses à simuler, en temps d'exécution, est le pipeline. Il représente l'élément central du coeur d'un processeur et intervient dans une large partie sur le comportement temporel de celui-ci. Notre but étant d'obtenir un simulateur rapide d'un processeur avec pipeline, nous avons déplacé la plupart du temps nécessaire aux calculs de la phase exécution à la phase compilation. Pour atteindre cet objectif, nous utilisons un automate à états finis pour le modèle interne du pipeline.

Ce chapitre traite de la modélisation des aspects architecturaux dans le cadre de la génération automatique d'un simulateur précis au cycle près. Nous allons utiliser l'expression *pipeline simple* pour désigner un pipeline de base, non composé de plusieurs branches (cas d'architecture superscalaire). Tout au long de ce chapitre, nous allons expliquer comment un pipeline simple est modélisé par un automate à états finis en donnant les algorithmes principaux et en introduisant les deux outils *p2a* et *a2cpp* développés au sein de l'équipe « Systèmes Temps Réel » par Mikaël Briday et Jean-Luc Béchennec. Ces deux outils permet de générer, à partir de la description du pipeline en *HARMLESS*, le code C++ correspondant.

Dans un premier temps, un état de l'art sur les différentes méthodes de modélisation d'un pipeline (automate d'états finis et réseau de Petri) sera fait.

Nous nous intéressons, dans un second temps, à la manière dont nous spécifions les contraintes à travers les *ressources* internes et externes et comment les temps d'exécution sont déduits de ces contraintes. Enfin, la façon dont nous modélisons le pipeline sous forme d'automate sera expliquée tout en présentant brièvement les deux outils cités plus haut.

## 5.2 État de l'art

La modélisation du matériel, plus particulièrement le pipeline, ainsi que l'ordonnancement des ressources ont été largement étudiés dans les ordonnanceurs d'instructions, qui sont utilisés par les compilateurs afin d'exploiter le parallélisme d'instructions (*ILP* pour *Instruction Level Parallelism*) dans le but de minimiser le temps d'exécution du programme. Ceci est obtenu en maximisant l'utilisation des ressources et en réduisant le nombre des bulles (suspensions) produites par les différents types d'aléas (aléas structurels, aléas de contrôle et de données) présentés dans la section 1.5.2.

### 5.2.1 Approche commune

Dans [26], Halambi et *al.* ont introduit une approche commune pour la modélisation du matériel basée sur une vue centrée sur le matériel. Dans cette approche, le processeur est modélisé par un ensemble de blocs fonctionnels. Ces blocks communiquent et sont synchronisés entre eux pour gérer les aléas. Un pipeline peut être modélisé par cette approche en modélisant un étage de pipeline par un bloc fonctionnel (par exemple, un module SystemC [33]). Cette approche est largement utilisée dans le processus de conception des processeurs, mais les simulateurs générés par ce type de modèles sont généralement lents à cause du coût de synchronisation de tous les blocs.

### 5.2.2 Utilisation d'un automate fini

Pour simuler temporellement le fonctionnement d'un processeur (le pipeline et les ressources en particulier), les travaux de Müller [41] reposent sur une modélisation à base d'automate fini et déterministe. Dans son travail, Müller est passé de l'approche classique de simulation à un modèle formel de processeur qui est le point de départ pour le développement de son algorithme.

#### 5.2.2.1 De l'approche classique au modèle formel proposé par Müller

L'approche classique [41] consiste à associer à chaque instruction un *template* comme le montre la figure 5.1 (reprise de [41]). Un template d'une instruction

montre pour chaque cycle d'horloge et après le lancement de cette instruction, quelles ressources (qui peuvent être une unité fonctionnelle ou un bus) seront utilisées. Donc, un template peut être assimilé à un tableau à deux dimensions où l'abcisse représentant le temps (en cycles d'horloge) et l'ordonnée les ressources utilisées par une instruction pour s'exécuter. Cocher l'une des cases du tableau indique que la ressource est utilisée par l'instruction et par suite occupée au cycle indiqué.

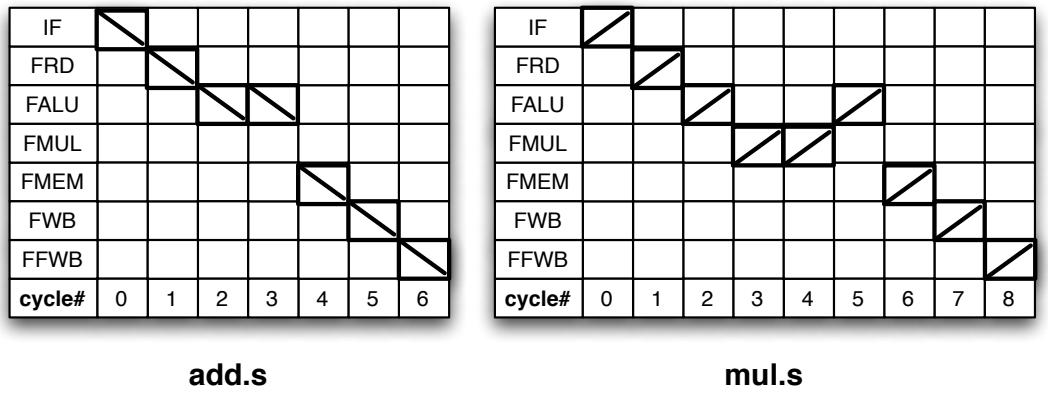
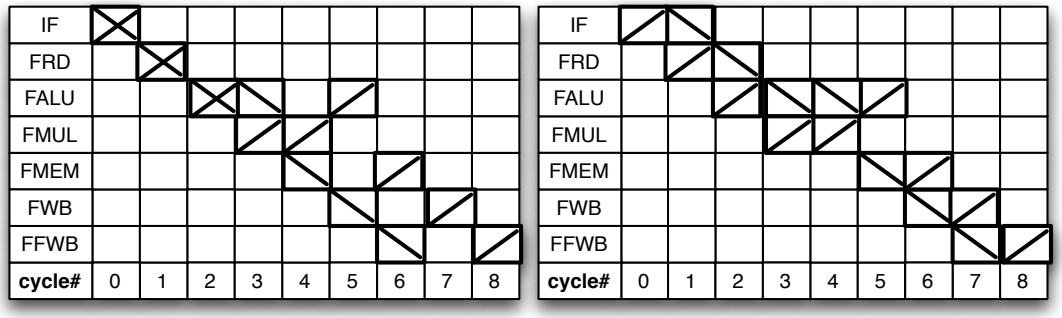



FIGURE 5.1 – Les templates pour les deux instructions *add.s* et *mul.s* du processeur *MIPS R 2010*.

Pour assurer la simulation de l'exécution d'une séquence d'instructions, un tableau, appelé table de réservation, ayant le même nombre de lignes que les templates est utilisé comme le montre la figure 5.2 (reprise de [41]). Ce tableau représente l'usage des ressources du processeur dans le futur par des instructions déjà lancées, ici pour les deux instructions *add.s* et *mul.s* du processeur *MIPS R 2010*. Au départ, la table de réservation est vide. Après le lancement de la première instruction, son template est fusionné dans la table de réservation (figure 5.2(a)). Après le lancement d'une seconde instruction, de la même manière, son template est mappé sur la table de réservation et s'il y a des conflits dans l'utilisation des ressources avec la première instruction, la deuxième instruction est décalée d'un cycle d'horloge (figure 5.2(b)). Sinon, elle peut être lancée, d'où le parallélisme des instructions et ainsi de suite.

D'après Müller, cette approche, bien que correspondant au modèle réel du processeur, conduit à un algorithme de simulation (présenté dans [41]) ayant mauvaises performances en temps de simulation (la table de réservation est construite dynamiquement), et nécessite (à cette époque) une grande quantité de mémoire pour stocker la table de réservation. Pour cela, Müller introduit, dans [41], le modèle




(a) Exécution simultannée de *add.s* et *mul.s*

(b) Début d'exécution de *mul.s* un cycle avant *add.s*

FIGURE 5.2 – Illustration des aléas entre les deux instructions *add.s* et *mul.s* du processeur *MIPS R 2010*.

formel du processeur qui est fortement lié au principe des tables de réservation présentées plus haut. Ce modèle permet de définir la fonction caractéristique du processeur qui relie chaque instruction du jeu d'instructions à un template, ainsi que l'opération de lancement d'une instruction. Il exprime le comportement temporel du processeur en modélisant le lancement d'une instruction d'une part et l'avance vers le cycle suivant d'autre part.

**Du modèle formel aux automates** Dans son approche, Müller a démontré que le processeur ne se bloque jamais et si une instruction est activée, elle sera exécutée après un nombre de cycles bien défini. Par conséquent, il est possible de définir un automate d'états finis déterministe qui simule l'exécution temporelle d'un processeur. Cet automate est défini par $(\Sigma, \Omega, \mathbb{N}, \phi, \delta, \varepsilon)$, avec :

- $\Sigma$ est l'ensemble d'états ;
- $\Omega$ est l'alphabet (jeu d'instructions du processeur) ;
- $\mathbb{N}$ est l'ensemble des états accepteurs (qui sont les états d'avancement de cycle) ;
- $\phi$ est la fonction de transition $(\Sigma \times \Omega \to \Sigma)$ ;
- $\delta$ est la fonction de sortie $(\Sigma \times \Omega \to \mathbb{N})$ ;
- $\varepsilon$ est l'état initial (table de réservation vide).

**Génération de l'automate** Comme nous l'avons déjà dit, Müller est parti du modèle formel pour développer son algorithme de génération de l'automate [41]. Cet algorithme se résume par : à partir d'une table de réservation vide $\{\varepsilon\}$, $\phi_\omega$ est appliquée à chaque instruction $\omega$ du jeu d'instructions et son template est ajouté

dans la table de réservation courante. Ce procédé est répété jusqu'à ce que toutes les possibilités d'activation d'instructions soient prises en comptes.

L'automate produit par cet algorithme peut devenir grand et contenir un degré élevé de redondance, il est minimisé ensuite en utilisant des méthodes standards de minimisation.

En ce qui concerne l'implémentation de cet algorithme, d'après Müller, la méthode la plus simple et la plus efficace est l'utilisation des tables bidimensionnelles représentant chaque fonction de transition et de sortie. Cependant les tables générées consomment rapidement beaucoup de mémoire.

Enfin, pour les processeurs ayant plusieurs unités fonctionnelles complètement indépendantes par rapport à l'usage des ressources, Müller a proposé de générer un automate pour chaque unité fonctionnelle au lieu d'un seul pour tout le processeur, ceci introduit un temps additionnel pour le simulateur, mais les tableaux deviennent beaucoup plus petits. Ces automates doivent être synchronisés afin que, par exemple, si l'on insère une bulle dans l'un, celle-ci soit également insérée dans les autres.

En conclusion, Müller propose d'utiliser un ou plusieurs automates à états finis pour modéliser le pipeline et construire un simulateur. Ensuite, le simulateur est utilisé par l'ordonnanceur d'instructions pour calculer le temps d'exécution des séquences d'instructions. Par contre, avec les processeurs modernes les automates peuvent être gros nécessitant l'utilisation des techniques de minimisation. De plus, Müller n'a considéré, dans sa modélisation, que les aléas structurels, les autres types d'aléas (aléas de données et aléas de contrôle) sont résolus en utilisant différentes heuristiques [14] qui peuvent être combinées avec son approche.

#### 5.2.2.2 Matrice de collision

Dans [47], Proebsting et Fraser utilisent la même approche, mais avec un algorithme différent qui produit directement un automate minimal. En effet, l'implémentation se base, elle aussi, sur des tables bidimensionnelles mais celles-ci ne représentent plus des tables de réservation mais des matrices de collision. Ils ont réutilisé l'automate DSTP [12] (pour *Davidson*, *Shar*, *Thomas* et *Patel*) mais simplifié.

En 1975, Davidson et *al.* (DSTP) ont proposé dans [12] d'utiliser les vecteurs de ressources pour générer un automate. Les vecteurs de ressources, comme les tables de réservation, permettent de détecter les aléas structurels.

L'automate DSTP est représenté par un seul tableau à trois dimensions. Il contient des entiers qui représentent la classe d'instructions $i$ (sachant qu'une classe d'instructions regroupe toutes les instructions qui utilisent les mêmes ressources au même cycle d'horloge), le compteur de cycle $c$ et l'état du pipeline $s$. Si le pipeline est dans l'état $s$, et si après $c$ cycles $i$ peut être activée sans provoquer

d'aléa structurel, alors l'automate produit un entier qui code un nouvel état du pipeline, sinon il annonce un aléa structurel.

L'algorithme DSTP modifié présenté dans [47] génère moins d'états. Cette fois ci, il représente chaque état $S$ par un tableau à deux dimensions à valeurs booléennes $S[I,t]$ qui est égale à 1, si et seulement si, l'instruction $I$ provoque un aléa structurel dans le cas où elle est activée $t$ cycles après que la machine entre dans l'état $S$ ; les états sont générés en utilisant les matrices de collision $M$ (tableau à trois dimensions à valeurs booléennes) qui enregistre quand les instructions se chevauchent. $M[I,J,t]$ est égale à 1, si et seulement si, l'activation de $J$, $t$ cycles après $I$, cause un aléas structurel, comme le montre l'exemple représenté par le tableau 5.1.

| **I** | **J** | **t** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
| add.s | add.s | 1 | 1 | 0 | 0 | 0 | 0 |
| | mov.s | 0 | 0 | 0 | 0 | 0 | 0 |
| | mul.s | 0 | 0 | 0 | 0 | 0 | 0 |

TABLE 5.1 – Ce tableau présente la matrice de collision de l'instruction d'addition *add.s*. Nous pourrons remarquer qu'une instruction *add.s* peut être suivie d'une instruction *mov.s* ou *mul.s* à n'importe quel instant, par contre une autre instruction *add.s* ne peut être activée qu'après écoulement d'au moins, deux cycles d'horloge.

l'automate est construit de la manière suivante [47] : à partir d'un état vide, représenté par une matrice nulle, toutes les instructions sont activées en produisant des nouveaux états. En effet, pour chaque état $S$, une instruction $I$ est activée si $S[I,1] = 0$. Et en utilisant la règle suivante : $S'[J,t] = S[J,t+1] \bigvee M[I,J,t]$, un nouveau état $S'$ est obtenu. Si $S'$ n'était pas créé avant, il est ajouté à l'automate comme une transition de $S$ à $S'$ en $I$. Cette construction se termine lorsque toutes les instructions possibles sont activées de tous les états possibles. Cet algorithme génère un automate à état fini minimal pour une machine donnée.

Comme conclusion, Proebsting et Fraser ont réussi à générer un automate plus rapide mais en tenant compte seulement des aléas structurels qui sont nécessaires dans les ordonnanceurs d'instructions.

#### 5.2.2.3 Automate direct, automate inverse et tables de jonction

Comme leurs prédécesseurs, Bala et Rubin expliquent dans [3] une technique pour détecter et éviter les aléas structurels dans un pipeline pendant la phase de

l'ordonnancement d'instructions du compilateur. Ils ont amélioré les algorithmes de [41] et [47] en permettant de remplacer les instructions dans des séquences déjà ordonnancées et ont présenté l'algorithme *BuildForwardFSA* qui est utilisé pour construire l'automate (voir [3]). De plus, ils ont présenté l'algorithme *BuildReverseFSA* qui génère l'automate inversé.

**L'algorithme *BuildForwardFSA*** Cet algorithme permet de générer un automate à états finis direct (par rapport à l'automate inverse, voir plus loin) (voir la figure 5.3). Comme dans les travaux de Proebsting et Fraser, chaque état de l'automate est représenté par une matrice de collision, le bit *i,j* dans cette matrice indiquant si l'instruction indexée par $i$ va créer un conflit avec l'état courant du pipeline, $j$ cycles dans le futur. Le premier état est une matrice nulle. Utilisant la même formulation, une classe d'instructions $C$ est représentée par une matrice de collision similaire. Dans ce cas, le bit *i,j* indique si une instruction de la classe d'instructions $i$ activée dans le cycle courant, va créer un aléa structurel avec cette classe d'instructions $C$ dans le cycle $j$. Les autres états de l'automate sont formés en faisant un *ou* entre la matrice de collision de l'état et celle de la classe d'instructions.

Pour comprendre le principe de cet algorithme, considérons une machine [3] dans laquelle, trois classes d'instructions sont présentes :

- $i$ : pour les instructions entières (*integer instructions*), comme `add`, `sub`,... et `inop` ;
- $f$ : pour les instructions flottantes (*floating point instructions*), comme `fadd`, `fsub`,... et `fnop` ;
- *ls* : pour les instructions de chargement/rangement (*load/store instructions*).

Supposons que trois ressources matérielles soient disponibles :

- la ressource de décodage des entiers *ID* ;
- la ressource de décodage des flottants *FD* ;
- la ressource d'accès mémoire *MEM*.

Si une instruction ne peut pas être activée, une des instructions *NOP* (pour *No OPeration*) va être activée. Les accès aux ressources par les différentes classes d'instructions sont donnés sur le tableau 5.2.

| classes d'instructions | numéro de cycle | |
|---|---|---|
| | 0 | 1 |
| i | ID | |
| f | FD | |
| ls | ID+MEM | MEM |

TABLE 5.2 – Ce tableau présente les accés aux ressources pour chaque classe d'instructions.

L'automate direct obtenu en utilisant l'algorithme *BuildForwardFSA* est donné sur la figure 5.3.

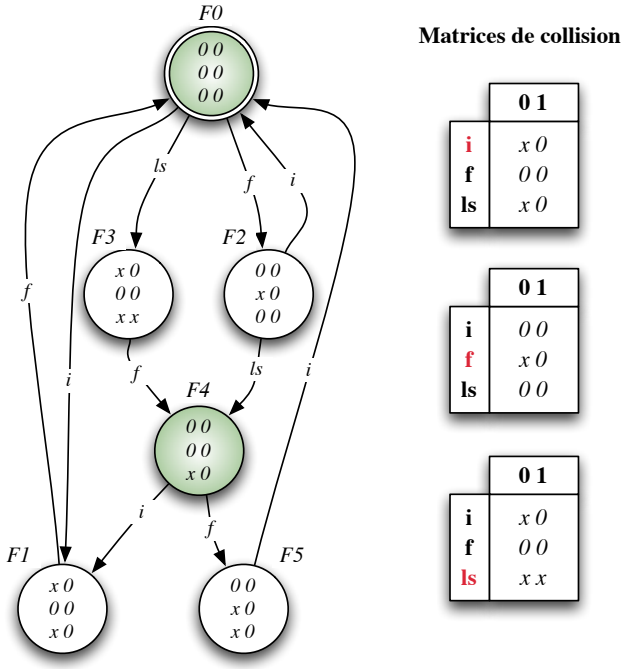



FIGURE 5.3 – Automate direct généré en utilisant l'algorithme *BuildForwardFSA*. Chaque état de l'automate est représenté par une matrice de collision et les transitions représentent le passage d'un état à un autre en activant une classe d'instructions (l'étiquette sur la flèche). La lettre $x$ permet d'exprimer les aléas structurels entre les différentes instructions. Les deux états *F0* et *F4* sont grisés car ce sont des états d'avancement (voir texte qui suit).

Sur cette figure, les matrices de collision de chaque classe d'instructions ainsi que les matrices de collision de chaque état sont représentées. Seules les transitions

qui ne provoquent pas des conflits (aléas structurels) sont présentes dans l'automate. L'état entouré par un double cercle est l'état de départ (matrice de collision nulle).

Prenons par exemple, l'état *F2*. À partir de cet état, une instruction de la classe d'instructions $i$ peut être activée (puisque dans la matrice de collision de cet état, il y a un zéro dans la première ligne, première colonne), en faisant un *ou* entre la matrice de collision de l'état *F2* et celle de la classe d'instructions $i$, une matrice où toute la première colonne est différente de zéro ($= x$) est obtenue, alors aucune instruction ne peut être activée de cet état et la matrice est avancée d'une colonne. Par suite, cet état *F0* est marqué et appelé état d'avancement d'un cycle (même raisonnement pour l'état *F4*).

Cet algorithme produit la table de transition de l'automate qui sera consultée par le compilateur quand il ordonnance les instructions.

L'algorithme de Proebsting et Fraser, utilisant les matrices de collision pour représenter les états, produit un automate minimal. Malgré tout, pour plusieurs machines réelles, l'automate reste toujours trop important. Pour résoudre ce problème, au lieu de construire un seul gros automate, Bala et Rubin propose une méthode de factorisation pour créer plusieurs petits automates dont le nombre total d'états est inférieur à celui d'origine. Cette méthode est basée sur le fait que généralement les implémentations des processeurs divisent l'ensemble d'instructions en différentes catégories, chaque catégorie étant exécutée par une unité fonctionnelle différente, et par suite le tableau d'utilisation des ressources peut être divisé en plusieurs tableaux à partir desquels les automates factorisés sont construits en utilisant l'algorithme *BuildForwardFSA*.

**Tables de jonction** Dans le but d'ordonnancer les instructions, l'automate obtenu à partir de l'algorithme *BuildForwardFSA* peut être démarré de n'importe quel état marqué comme un état d'avancement d'un cycle. Cependant, pour un bloc d'instructions ayant plusieurs prédécesseurs déjà ordonnancés, un problème survient lors du choix de cet état d'avancement. Pour résoudre ce problème, Bala et Robin ont modifié l'algorithme *BuildForwardFSA* de telle façon qu'à chaque fois qu'un nouveau état d'avancement d'un cycle est obtenu, un *ou* est fait entre sa matrice de collision et la matrice de collision de chacun des autres états d'avancement d'un cycle créé avant, ce qui aboutit à des états appelés états de jonction qui ne correspondent pas forcément à des états présents dans l'automate d'origine. Chaque état de jonction est noté par $Join(Fi, Fj)$ pour un couple donné d'états d'avancement d'un cycle $Fi$ et $Fj$. Les nouveaux états de jonction sont reconnaissables par le fait qu'ils n'ont pas de transitions en entrée mais seulement des transitions de sorties. Afin de pouvoir choisir l'état de jonction approprié, une table de jonction représentant tous les couples d'états d'avancement d'un cycle est

créée. Cette table est une matrice carrée (par définition) et symétrique (commutativité de l'union). Seule la partie triangulaire de la matrice présente le résultat, puisque la jonction d'un état d'avancement de cycle avec lui-même n'a pas vraiment d'intérêt. La table de jonction pour l'automate de la figure 5.3 est donnée par le tableau 5.3.

| | **F0** | **F4** |
|---|---|---|
| **F0** | F0 | F4 |
| **F4** | F4 | F4 |

TABLE 5.3 – Ce tableau présente la table de jonction de l'automate direct de la figure 5.3. Par exemple, la matrice de collision pour $Join(F0, F4) = F4$

**L'algorithme *BuildReverseFSA*** L'algorithme *BuildForwardFSA* permet de créer un automate direct. Cependant, si les transitions de cet automate sont inversées, un automate complémentaire est obtenu et appelé automate inverse. Ce type d'automate permet de détecter les aléas non seulement au cycle courant mais aussi dans les cycles futurs.

L'automate direct est déterministe (chaque état a au plus une transition de sortie pour une étiquette donnée), mais s'il est inversé pour produire l'automate inverse, ce dernier a de fortes chances de ne pas l'être (puisque dans l'automate directe, certains états ont des transitions d'entrée possédant des mêmes étiquettes). Pour lutter contre ce problème, de tels états sont groupés ensembles pour former un état dans l'automate inverse. L'algorithme *BuildReverseFSA* présenté dans [3] crée directement l'automate inverse déterministe pour un automate direct donné. En appliquant cet algorithme, l'automate inverse, obtenu à partir de l'automate direct de la figure 5.3, est donné sur la figure 5.4. Dans ce cas, un état représente un ensemble d'états de l'automate direct correspondant et non une matrice de collision et les transitions représentent le passage d'un état à un autre en activant une classe d'instructions (l'étiquette sur la flèche).

Enfin, la combinaison des deux automates direct et inverse forme un outil efficace pour les ordonnanceurs d'instructions. Une des applications importantes de cet outil est qu'il autorise la modification d'une séquence d'instructions (en remplaçant une instruction au milieu par exemple) même après l'avoir ordonnancée, à condition que ce changement ne crée pas un aléa structurel et en considérant qu'il ne viole aucune des dépendances de données.

Comme leurs prédécesseurs, Bala et Rubin, dans leur approche, n'ont pris en compte que les aléas structurels.

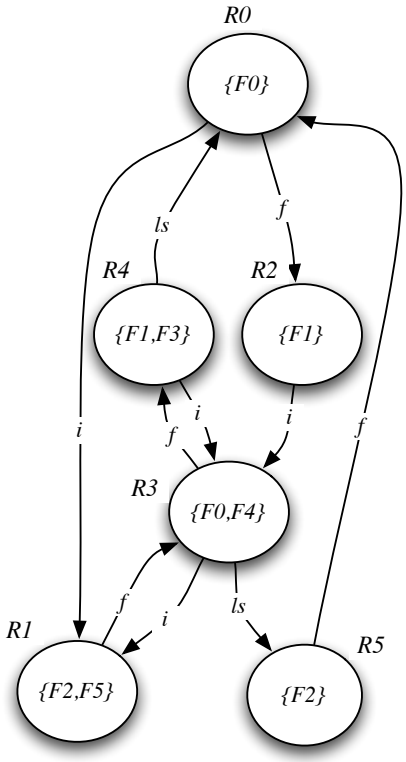



FIGURE 5.4 – Automate inverse généré en utilisant l'algorithme *BuildReverseFSA*.

### 5.2.3 Une autre approche : Réseaux de Petri

Une autre approche pour modéliser et analyser le fonctionnement d'un pipeline consiste à utiliser les réseaux de Petri. Un réseau de Petri est formé d'un ensemble de places, de transitions et d'arcs les reliant les uns aux autres. Donc, c'est un graphe biparti (c'est-à-dire composé de deux types de nœuds) orienté. Les places peuvent contenir des jetons qui circulent de place en place en franchissant des transitions et représentent généralement des ressources disponibles. Les modèles obtenus en utilisant les réseaux de Petri sont souples, puissants, faciles à visualiser (pour un modèle simple) et permettent en outre de nombreuses analyses formelles (voir section 1.2.1.1). Ils ont donc été utilisés avec succès dans la modélisation des processeurs pipelinés.

#### 5.2.3.1 Les réseaux de Petri classiques

Les réseaux de Petri classiques (non colorés, non temporisés, ...) ne peuvent fournir qu'une analyse qualitative du flux d'instructions permettant ainsi la vérification de quelques propriétés comme la détection des interblocages. Dans le cas d'une analyse temporelle (quantitative) du fonctionnement d'un processeur, ils s'avèrent insuffisants puisque chaque instruction doit être définie avec précision, non seulement par son type et ses opérandes mais aussi par ses dépendances, ses cibles et ses branchements, etc., qui affectent son profil temporel dans le flux d'instructions.

#### 5.2.3.2 Utilisation des réseaux de Petri temporisés

Dans [51], Razouk décrit un ensemble d'outils (*P-NUT system* pour *Petri Net Utility Tools system*) permettant la simulation, l'animation ainsi que l'analyse du comportement des modèles construits en utilisant les réseaux de Petri temporisés (en particulier l'évaluation des performances et l'analyse temporelle des microprocesseurs à pipeline) où (voir la figure 5.5) :

- les *places* contiennent des jetons représentant les instructions d'une part, ou permettant d'indiquer la disponibilité d'un bus ou des places dans un cache d'instructions par exemple, d'autre part ;
- les *transitions* représentent des événements comme le décodage, la lecture des opérandes, ... ;
- les *arcs* peuvent être associés avec un poids (nombre) permettant d'exprimer, par exemple, le nombre maximal d'instructions récupéré simultanément depuis la cache d'instructions ;
- le *temps* permet de modéliser le fait qu'un événement nécessite un temps (en nombre de cycles d'horloge) pour se réaliser.

Le principe de fonctionnement de ces outils est le suivant : un simulateur appelé *P-NUT simulator* [51] prend en entrée un réseau de Petri temporisé modélisant un processeur à pipeline, par exemple, et quelques commandes de simulation qui autorisent l'utilisateur à contrôler le temps de simulation. La simulation fait circuler les jetons (les instructions) dans ce réseau, et fournit en sortie, une trace, décrivant la façon dont les états du système changent avec le temps. Cette trace est ensuite utilisée par un outil d'analyse. Dans cet ensemble d'outils, les modèles obtenus sont statistiques (approche non déterministe) et ne peuvent pas être utilisés pour des simulations détaillées. Ceci est dû au fait que l'utilisateur donne une probabilité pour qu'une instruction assure une fonction donnée, comme l'écriture des résultats dans les registres, le nombre de cycles nécessaire à son exécution... Par exemple, l'utilisateur spécifie que l'exécution d'une instruction nécessite $1, 2, 5$ cycles d'horloges avec des probabilités respectives de $0.5, 0.3, 0.2$. De plus, Razouk

modélise chaque étage de pipeline avec un réseau de Petri temporisé différent, d'où la nécessité de synchronisation entre les différents réseaux de Petri, et dans le cas d'un long pipeline la simulation risque d'être beaucoup plus lente. Par exemple, la figure 5.5 montre le réseau de Petri temporisé de l'étage *Decode* d'un pipeline (décodage de l'instruction et la lecture de ses opérandes). Sur cette figure, nous pouvons remarquer que le décodage d'une instruction nécessite un cycle d'horloge (spécifié par *Firing time*). Le temps d'attente avant le tir de la transition est donné par *enable time*, ici égal à 0.

Enfin, dans ce type de modélisation, le comportement fonctionnel n'est pas pris en compte. Le simulateur ne peut pas exécuter un code binaire réel.

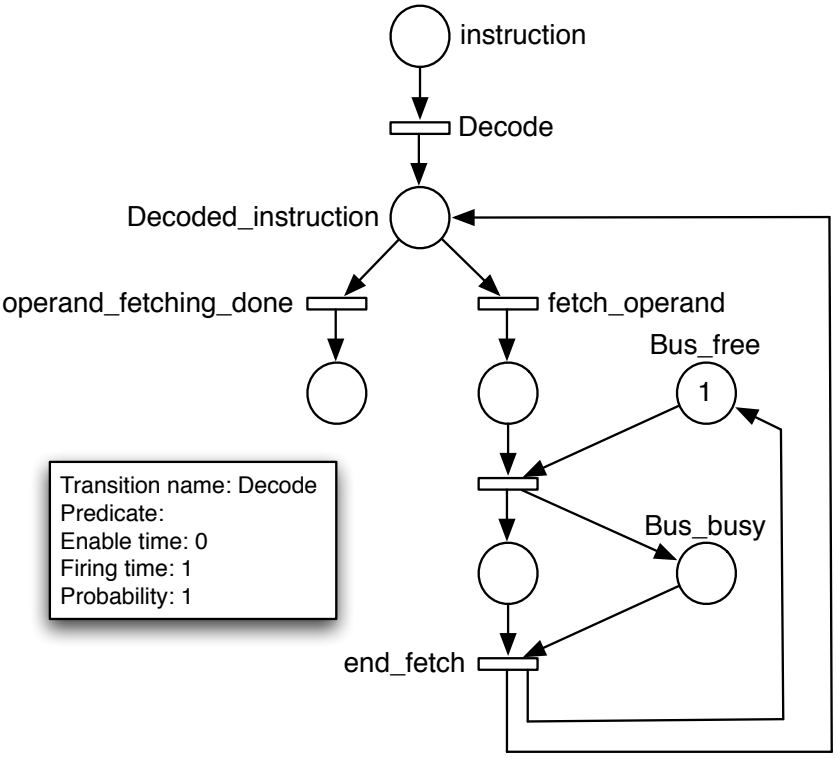



FIGURE 5.5 – Réseau de Petri représentant l'étape de décodage de l'instruction ainsi que la lecture de ses opérandes (étage *Decode* d'un pipeline). Les places vont contenir des jetons représentant les instructions et les transitions représentent des événements.

#### 5.2.3.3 Utilisation des réseaux de Petri colorés (*CPN*)

Comme les réseaux de Petri colorés [31] offrent de plus la possibilité d'attribuer des types de données complexes aux jetons, ils sont capables de modéliser le comportement temporel d'un processeur. C'est une technique de modélisation souple permettant la description du parallélisme, aussi bien que la synchronisation et le partage des ressources.

Dans [8] et [7], Burns et *al.* ont développés un ensemble d'outils logiciels associé à une méthodologie leur permettant de modéliser et analyser les caractéristiques temporelles (comme par exemple le *WCET*) des plates-formes matérielles en se basant sur les réseaux de Petri colorés (*Design/CPN tool*). Ils se sont intéressés surtout à la modélisation des processeurs asynchrones superscalaires.

Dans leur modèle de processeur :

- les différents types d'instructions sont spécifiés avec des identifiants prédéfinis (comme par exemple, *BRA* pour définir les opérations de branchement, *INT* les opérations sur les entiers,...), qui sont modélisés par des *jetons auxquels des données complexes peuvent être attachées*, telles que le numéro de l'instruction dans le flot d'instructions à exécuter, son type, ses dépendances de données, etc. ;
- le flot d'instructions est déterminé par des *gardes* qui sont des conditions attachées aux *arcs* du modèle et qui détermine si une transition a le droit de tirer ou non un jeton donné. En d'autres termes, les gardes permettent de sélectionner les instructions à exécuter et de détecter les aléas de données. Ainsi, ils déterminent le comportement dynamique du modèle ;
- le pipeline est modélisé par un ensemble de *transitions*. Chaque transition définit un étage du pipeline ou une unité d'exécution. Entre les transitions, les *places* sont utilisées pour le stockage, les files d'attente, etc. La détection et la gestion des aléas structurels se modélisent par l'utilisation des *boucles de rétroaction* des unités fonctionnelles indiquant leur disponibilité.

De plus, leur méthode permet de modéliser la prédiction de branchement, l'exécution dans le désordre, les caches,... Mais comme dans l'approche précédente [51], le comportement fonctionnel n'est pas pris en compte ce qui rend impossible l'exécution d'un code binaire réel, et le modèle obtenu pour un processeur complexe semble lent à exécuter et devient vite peu compréhensible.

**Utilisation des réseaux de Petri colorés réduits (*RCPN*) [52]** Comme nous venons de le présenter, les *CPN* ont été utilisés pour décrire le comportement temporel d'un processeur. Toutefois, pour des processeurs réels, les modèles obtenus sont très complexes. Ceci est essentiellement dû au peu d'efficacité du mécanisme à base de jeton pour capturer et résoudre les aléas de données. Pour améliorer la performance du modèle obtenu avec les *CPN*, Reshadi et *al.* présentent

dans [52] les réseaux de Petri colorés réduits (*RCPN*). C'est une approche de modélisation générique permettant la génération des simulateurs *CAS* pour les processeurs pipelinés d'une manière formelle et concise.

Les *RCPN* héritent de toutes les caractéristiques de *CPN*, tout en redéfinissant un certain nombre de leur concepts de la manière suivante :

- Les places : Dans ce modèle, une place montre l'état d'une instruction. Un étage de pipeline est assigné à une ou plusieurs places, il possède un paramètre de capacité qui indique combien de jetons (instructions) peuvent être présents en même temps dans cet étage. Ce paramètre de capacité est commun à toutes les places représentant le même étage du pipeline ;
- Les transitions : Une transition représente la fonctionnalité qui doit être exécutée lorsque l'instruction modifie son état (place). Cette fonctionnalité est exécutée lorsque la transition est activée, c'est-à-dire lorsque son état de garde est vrai ; de plus il y a assez de jetons de type approprié sur l'arc d'entrée, et les étages de pipeline des places de sortie ont une capacité suffisante pour accepter de nouveaux jetons. Pour contrôler le flot d'instructions, une transition peut accéder directement aux autres unités du processeur telles que la prédiction de branchements, la mémoire cache, etc., qui sont définis dans une bibliothèque à part ;
- Les arcs : Un arc peut avoir une expression qui teste par exemple le type ou la disponibilité des registres sources de l'ensemble des jetons qui passent à travers cet arc. Pour l'exécution déterministe, chaque arc de sortie d'une place a une priorité qui donne l'ordre au cours duquel les transitions correspondantes sont activées et peuvent consommer les jetons. Les priorités à l'arc sont ajoutées au modèle afin de simplifier les conditions de garde des transitions et les rendre statiques ;
- Les jetons : Deux types de jetons peuvent être présents dans une place : les jetons *instruction* qui transportent, selon le type de cette instruction, des données, et les jetons de *réservation* qui ne transportent pas de données, mais leur présence dans une place indique le taux d'occupation de cette place. Les instructions, qui ont des comportements similaires, sont groupées ensemble ce qui permet d'améliorer la productivité et l'efficacité du modèle.

Pour résoudre les aléas de données, les *RCPN* utilisent une approche alternative basée sur les sémaphores (mécanisme de blocage/déblocage) qui contrôlent l'accès aux registres.

Pour résumer, chaque jeton instruction représente une instance d'une instruction en cours d'exécution dans le pipeline. Le *RCPN* décrit comment une instruction circule à travers les différents étages du pipeline. En d'autres termes, pour chaque type d'instructions, il décrit le prochain étage de pipeline où le jeton instruction va passer, quand l'instruction peut avancer en franchissant une transition,

et ce qui devrait être fait durant cette transition. Dans chaque *RCPN*, il y a un sous-réseau indépendant (voir figure 5.6(c)) qui permet de générer les jetons instruction, et pour chaque type d'instructions, il y a un sous-réseau correspondant qui décrit distinctement le comportement des jetons *instruction* de ce type pour chaque étage de pipeline. Ainsi, une fois l'instruction décodée [53], son comportement dans le temps (cycle d'horloge) et l'espace (étage du pipeline) sera connu (approche statique qui permet d'accélérer la simulation).

La figure 5.6, prise de [52], permet de mieux comprendre le modèle proposé par Reshadi et *al.*. Cette figure représente la structure d'un pipeline simple (figure 5.6(a)) composé de 4 unités (*U1*, *U2*, *U3* et *U4*) et de deux tampons (*L1* et *L2*) ainsi que son modèle obtenu en utilisant un *CPN* (figure 5.6(b)) d'une part et un *RCPN* (figure 5.6(c)) d'autre part. Dans la figure 5.6(b), les places (états) sont représentées par des cercles, les transitions (fonctions) par des barres horizontales et les points représentent les jetons. Quand un jeton est présent dans la place *L1*, cela veut dire que dans la figure 5.6(a) le tampon *L1* est vide et peut recevoir une instruction de l'unité *U1*. Lorsqu'un jeton arrive dans la place *P1*, cela est interprété au niveau du pipeline par la présence d'une instruction dans le tampon *L1*, prête d'être traitée soit par l'unité *U2* soit par l'unité *U4*. Sur la figure 5.6(c), le modèle est divisé en 3 sous-réseaux (*S1*, *S2* et *S3*). *S1* représente le sous-réseau indépendant permettant de générer les jetons pour deux types d'instructions, *S2* et *S3* décrit distinctement le comportement des jetons *instruction* de chaque type pour chaque étage de pipeline.

Enfin, le modèle obtenu avec les *RCPN* est aussi souple que celui obtenu en utilisant les *CPN*, mais beaucoup moins complexe. Il permet de générer un simulateur précis au cycle près. Pour chaque sous-réseau dépendant d'un type d'instructions, une classe template *C++* est générée. Cette classe sauvegarde les valeurs des différents symboles et possèdent une fonction décrivant le comportement de l'instruction pour une place donnée (étage du pipeline). Donc, dans ce modèle, l'exécution de chaque instruction est divisée en plusieurs parties, et à chaque cycle d'horloge l'instruction passe d'un étage à un autre tout en exécutant le code correspondant, ce qui a tendance à ralentir un peu la simulation surtout pour un processeur complexe.

### 5.2.4 Adéquation de ces travaux par rapport à notre objectif

Notre but est d'obtenir un simulateur très rapide et précis au cycle près du processeur contenant un pipeline, et pour ce faire nous essayons de déplacer la plupart du temps nécessaire aux calculs, de la phase exécution à la phase compilation. Pour cela, nous avons choisi d'utiliser un automate.

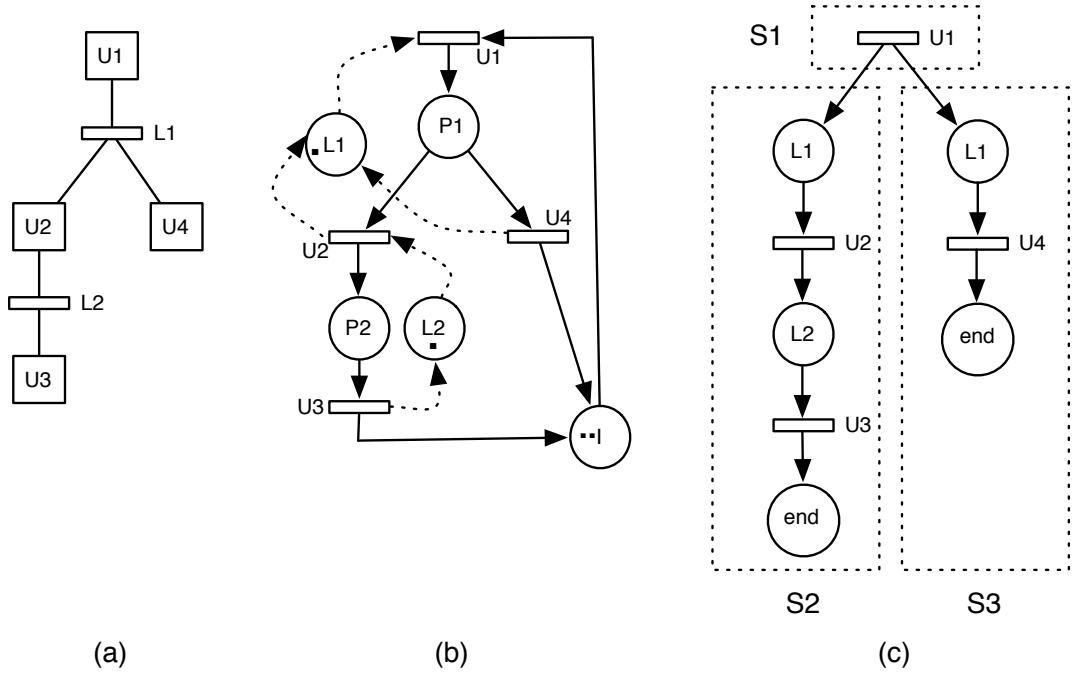



FIGURE 5.6 – (a) Structure d'un pipeline, (b) Son modèle *CPN* et (c) Son modèle *RCPN*.

Dans les travaux se basant sur des automates, seuls les aléas structurels ont été pris en compte ce qui est logique pour les ordonnanceurs d'instructions. Dans notre travail présenté ci-après, nous étendons l'algorithme *BuildForwardFSA* pour prendre en compte les aléas de données, les aléas de contrôle et les ressources partagées (comme par exemple les timers, les contrôleurs de mémoire, ...) qui peuvent être détenues par des périphériques externes qui ne sont pas modélisés à l'aide d'un automate. Chaque état de l'automate représente un état de pipeline à un instant donné pour lequel tous les aléas ont déjà été résolus, à part les ressources partagées qui seront résolues dynamiquement. Cet automate est obtenu dans la phase compilation. Et comme dans notre modèle l'exécution d'une instruction se fait totalement avant de passer dans le pipeline (une seule fonction d'exécution, voir section 7.5), la simulation temporelle consiste à parcourir l'automate selon les instructions du programme avec pour conséquence une simulation très rapide.

## 5.3 Notre modélisation du pipeline

Dans notre cas, nous nous intéressons à modéliser un pipeline (c'est-à-dire les étages et les différentes catégories d'aléas) sous forme d'automate à états finis déterministe.

En effet, il existe plusieurs méthodes pour modéliser un pipeline (voir section 5.2), mais nous avons choisi la modélisation sous forme d'automate à états finis déterministe pour plusieurs raisons :

- certaines contraintes sont résolues au moment de la génération de l'automate, c'est-à-dire au moment de la génération du simulateur. Cette propriété intéressante signifie que des calculs sont effectués une fois pour toute, sans ralentir ensuite la simulation ;
- la transformation d'un automate en code exécutable est particulièrement efficace. Au niveau du code généré, passer d'un état à un autre revient à lire une valeur dans une table, ce qui est très rapide, contrairement à d'autres modèles comme les réseaux de Petri ;
- le temps d'accès au prochain état de l'automate est en temps constant, et ainsi la vitesse de simulation de l'automate n'est pas dépendante du nombre d'états de l'automate. Ceci reste toutefois à relativiser avec l'utilisation de la mémoire cache de la machine hôte.

Dans notre approche, un état de l'automate représente un état de pipeline à un instant $t$ comme l'illustre la figure 5.7. Les transitions correspondent à une avancée d'un cycle d'horloge et permettent donc de passer de l'état du pipeline à l'instant $t$ à son état à l'instant $t + 1$. L'automate représente alors l'ensemble des scenarios possibles d'exécutions.

À chaque cycle d'horloge, le pipeline passe d'un état à un autre en prenant en compte tous les types d'aléas (structurels, de données et de contrôle). Les aléas de contrôle sont résolus pendant la simulation : si le branchement est pris, les instructions qui sont exécutées à la suite d'une instruction de branchement sont dynamiquement remplacés par des instructions *NOP*. Les contraintes résultant des insuffisances structurelles et les aléas de données sont prises en compte à la génération de l'automate, et modélisées en utilisant les ressources.

### 5.3.1 Les ressources

Les ressources sont définies comme un mécanisme permettant de décrire les dépendances et les concurrences d'accès aux composants dans un pipeline (le calcul du temps en résulte). Elles sont utilisées pour prendre en compte les aléas structurels et les aléas de données dans le pipeline. Deux types de ressources sont définis, les ressources internes et les ressources externes, modélisant les différentes contraintes respectivement statiquement (résolues pendant la phase de génération

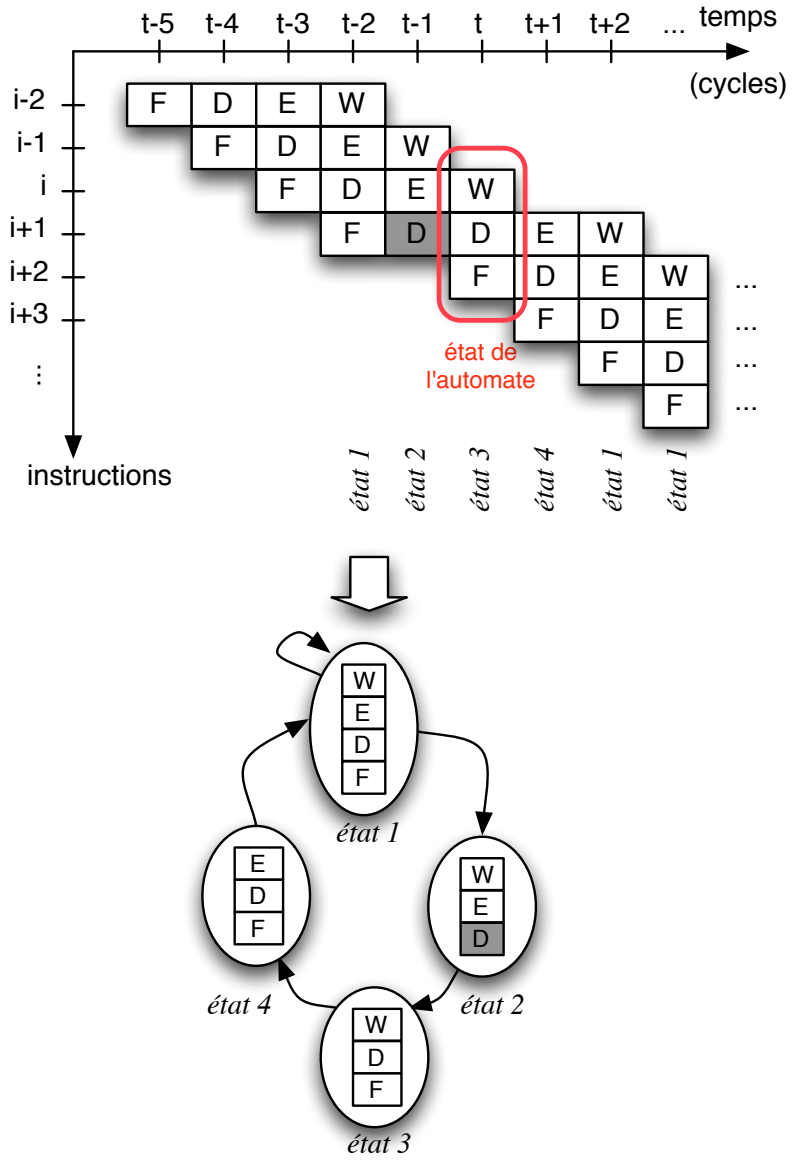



FIGURE 5.7 – Un état de l'automate représente un état du pipeline à un instant donné. Dans cet exemple, avec un pipeline à 4 étages, 3 instructions sont dans le pipeline à l'instant t, et l'étage 'D' est bloqué à l'instant t-1. L'automate modélise la séquence du pipeline, en considérant qu'il n'y a qu'un seul type d'instructions (pour une raison de clarté).

de l'automate) et dynamiquement (résolues pendant la simulation).

#### 5.3.1.1 Les ressources internes

Les ressources internes peuvent être comparées aux ressources définies par Müller dans [41]. Elles sont principalement utilisées pour modéliser les aléas structurels. Comme le champ de ces ressources est limité aux contraintes qui sont complètement sous le contrôle du pipeline, seules les instructions qui sont dans le pipeline peuvent *prendre* ou *relâcher* une ressource interne.

Quand l'état du pipeline est connu (c'est-à-dire les instructions présentes dans chaque étage de pipeline sont définies), alors l'état de chaque ressource interne est complètement défini (*prise* ou *relâchée*). Dans ce cas, lorsque l'automate est construit (et puis le simulateur), les contraintes décrites par les ressources internes sont directement résolues lorsque l'ensemble des états suivant est construit. De ce fait, ces ressources sont prises en compte au moment de la génération de l'automate (approche statique), et aucun supplément de calcul n'est requis pour vérifier ce type de contrainte pendant la simulation.

Par exemple, chaque étage de pipeline est modélisé sous la forme d'une ressource interne. Dès qu'une instruction entre dans un étage, elle prend la *ressource interne* associée. Elle relâche cette ressource lorsqu'elle quitte l'étage. La contrainte ajoutée dans notre description du pipeline est alors que *chaque étage de pipeline peut contenir au plus une seule instruction.*

#### 5.3.1.2 Les ressources externes

Les ressources externes sont utilisées pour décrire les ressources qui sont *partagées* avec d'autres composants matériels qui évoluent concurremment, comme un *timer* ou un contrôleur mémoire. Les ressources externes sont alors une extension des *ressources internes* pour décrire des *contraintes dynamiques* qui seront résolues pendant la simulation. Par exemple, avec un contrôleur de mémoire, le pipeline est bloqué s'il fait une requête au contrôleur et que celui-ci est occupé (par un autre processeur par exemple, ou un *DMA* (*Direct Memory Access*) - accès direct à la mémoire -, ...). Si le contrôleur mémoire est disponible, l'étage de pipeline qui a fait la requête peut alors prendre la ressource.

Si une ressource externe peut être prise par une instruction dans plus d'un étage de pipeline, une priorité est mise en place entre les étages. Par exemple, deux étages de pipeline peuvent être en compétition pour un accès mémoire sur un microcontrôleur avec un pipeline simple, sans cache de données et d'instructions.

Une propriété intéressante des ressources externes est qu'elles permettent de contrôler les aléas de données. Dans ce cas, une ressource externe est utilisée,

associée à un contrôleur de dépendance de données[1]. Ce contrôleur permet de contrôler les accès (en écriture et en lecture) aux registres, et ceci en positionnant la ressource externe correspondante à *disponible* ou *non disponible* selon que les registres sources nécessaires à une instruction pour pouvoir avancer dans le pipeline seront mis à jour ou non par d'autres instructions déjà en exécution dans le pipeline.

### 5.3.2 Les classes d'instructions

Dans la description du pipeline, les instructions peuvent être regroupées suivant leur comportement temporel. Celui-ci est directement lié aux ressources qui vont être prises et relâchées au cours de l'exécution de ces instructions dans les différents étages du pipeline. Ainsi, pour réduire l'espace d'état de l'automate, les instructions qui utilisent les *mêmes ressources* (internes et externes) sont regroupées pour former des *classes d'instructions*.

Le nombre de classes d'instructions est limité à $2^{R_{ext}+R_{int}}$ ($R_{ext}$ et $R_{int}$ sont respectivement le nombre de ressources externes et internes dans le système), mais ce maximum n'est pas atteint parce que certaines ressources sont partagées par l'ensemble d'instructions, comme les étages du pipeline, ce qui conduit à obtenir moins de classes d'instructions.

### 5.3.3 Les classes d'instructions de dépendance de données

De la même manière, les instructions qui font les mêmes accès en lecture et en écriture au banc de registres dans les mêmes étages du pipeline sont groupées dans des classes d'instructions de dépendance de données. À noter que ces classes ne jouent aucun rôle dans la génération de l'automate, elles seront surtout utilisées pour pouvoir générer des fonctions permettant, à partir de l'interrogation du contrôleur de dépendance de données, de savoir si la ressource externe correspondante est disponible ou non.

### 5.3.4 Génération de l'automate d'états finis

L'automate représente l'ensemble de scenarios possibles d'exécution des instructions dans le pipeline. Dans notre approche, nous étendons l'algorithme *Build-ForwardFSA* ([3]) pour prendre en compte toutes les catégories d'aléas (les aléas structurels, de données et de contrôle).

1. Le contrôleur de dépendance de données sera expliqué plus en détails dans la section 7.4.

#### 5.3.4.1 Les états

Un état de l'automate représente l'état du pipeline à un cycle donné et est défini par la liste de toutes les paires *(classe d'instructions, étage de pipeline)* dans le pipeline à un instant donné. Pour un système avec $c$ classes d'instructions, il y a alors $c + 1$ cas possibles pour chaque étage $e$ de pipeline (chaque classe d'instructions, ou une bulle). L'automate est alors fini, car il y a au plus $(c + 1)^e$ états. L'état initial est celui qui représente un pipeline vide.

Par exemple, sur un modèle basé sur le jeu d'instructions du co-processeur *RISC XGate* du *HCS12X* de *FreeScale* [19], avec un pipeline fictif (car il n'y a en pas en réalité) de 6 étages (voir section 5.4.2), 6 classes d'instructions sont nécessaires (il y a 4 ressources externes), ce qui conduit donc au plus à 117 649 états pour l'automate.

#### 5.3.4.2 Les transitions

Une transition est franchie à chaque cycle d'horloge. Elle dépend uniquement de :

- l'état des ressources externes (disponibles ou non) ;
- la classe de la prochaine instruction qui va entrer dans le pipeline.

Un intérêt majeur de cette approche réside dans le fait que les ressources internes, et à cause de leur caractère statique, sont résolues directement au niveau de la génération de l'automate. Par conséquent, elles n'apparaissent pas dans l'évaluation de la transition. Cette condition de transition (état des ressources externes et prochaine classe d'instructions) est appelée *condition de transition basique*. Comme plusieurs conditions peuvent apparaître pour aller d'un état à un autre, la condition de transition globale est alors une disjonction des conditions de transitions basiques.

Le nombre total de transitions est limité au maximum à $c \times 2^{R_{ext}}$ pour chaque état (où $R_{ext}$ est le nombre de ressources externes dans le modèle). Il y a donc au plus $(c + 2)^e \times 2^{R_{ext}}$ transitions pour l'automate entier.

Pour notre exemple basé sur le co-processeur *XGate* avec une architecture à 6 étages, avec 6 classes d'instructions et 4 ressources externes, nous obtenons alors 96 transitions possibles par états ($6 \times 2^4$). Pour l'automate entier, il y a donc au plus 4 194 304 transitions ($(6 + 2)^6 \times 2^4$).

#### 5.3.4.3 Algorithme de génération de l'automate

Les deux outils qui nous permettent de générer un simulateur modélisant le fonctionnement d'un pipeline simple sont *p2a* et *a2cpp* (voir figure 5.8).

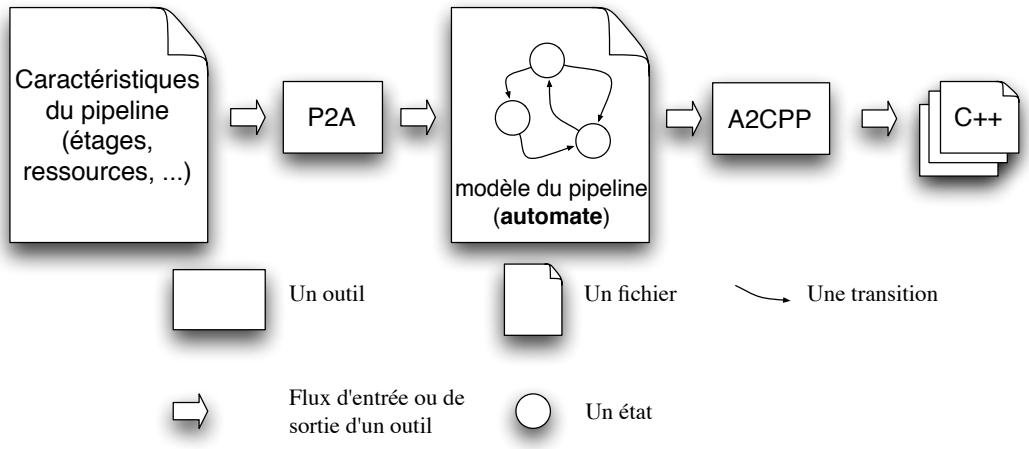



FIGURE 5.8 – Chaîne de développement d'un simulateur du pipeline simple.

L'outil *p2a* génère un automate à états finis à partir d'un fichier contenant des informations sur le pipeline à modéliser. Les informations concernent les étages, les différentes classes d'instructions, les ressources internes et externes : par quelles classes d'instructions elles sont utilisées et dans quel étage. L'outil *a2cpp* permet de générer le code *C++* correspondant. Un exemple de ce fichier est donné dans l'annexe C.

L'algorithme de génération de l'automate à états finis est présenté dans l'algorithme 1. Il est basé sur l'algorithme d'exploration en largeur des graphes afin d'éviter les problèmes de pile. L'idée de base de cet algorithme est qu'à partir de l'état initial, il calcule toutes les conditions de transitions basiques possibles. De l'état courant, chaque transition possible est prise pour obtenir l'ensemble des états suivants de l'automate. L'algorithme est ensuite réitéré pour chaque état qui n'a pas été encore traité.

- Créer une liste *FIFO* contenant l'état initial de l'automate ;
- Créer un automate, avec l'état initial ;
**Tant que** (La liste est non vide) **faire**
  - Prendre un état d'automate de la liste (état de départ) ;
  - Générer toutes les conditions basiques possibles (état de ressources externes combiné à la classe d'instructions de l'instruction suivante) ;
  **Pour** Chaque condition basique **faire**
    - *Construire l'état suivant de l'automate* (c'est un automate déterministe), utilisant la condition basique et l'état de départ ;
    **Si** (L'état n'est pas encore inclu dans l'automate) **Alors**
      - Ajouter le nouveau état (état cible) à la fin de la liste ;
      - Ajouter le nouveau état dans l'automate ;
    **Fin Si**
    **Si** (La transition n'existe pas) **Alors**
      - Créer une transition avec une condition basique vide ;
    **Fin Si**
    - Mettre à jour la condition de la transition, en ajoutant une condition basique (disjonction) ;
  **Fin Pour**
  - Supprimer l'état de départ de la liste ;
**Fin Tant que**

Algorithme 1: Génération de l'automate d'états finis, modèle du pipeline.

La fonction centrale de cet algorithme (notée « Construire l'état suivant de l'automate » dans l'algorithme 1) est celle qui permet d'obtenir l'état suivant de l'automate, quand une condition basique est connue. À partir d'un modèle générique de pipeline, cette fonction calcule le prochain état d'un automate, en tenant compte de toutes les contraintes générées par les ressources (internes et externes). Un pipeline est modélisé par une liste ordonnée contenant les étages du pipeline, où chaque étage est une ressource interne. L'algorithme 2 donne une version simplifiée de cette fonction. Dans la boucle de cet algorithme, nous remarquons que les étages du pipeline sont extraits du dernier au premier, et ceci parce que l'étage du pipeline qui suit celui en cours doit être vide pour recevoir une nouvelle instruction.

De plus, cette fonction permet de détecter les états causant un interblocage (*deadlock*, en anglais) dans l'automate (non représenté dans l'algorithme 2, pour des raisons de clarté). Ce type d'état survient lors d'une description erronée de pipeline à modéliser.

**Pour** Chaque étage du pipeline, du dernier étage au premier **faire**
  **Si** (Il existe une classe d'instructions dans l'étage courant) **Alors**
    **Si** (Les ressources requises par la classe d'instructions peuvent être prises dans l'étage suivant du pipeline) **Alors**
      - La classe d'instructions libère les ressources dans l'étage courant du pipeline ;
      **Si** (Il y a un étage suivant dans le pipeline) **Alors**
        - La classe d'instructions est déplacée vers l'étage suivant du pipeline ;
        - Les ressources nécessaires dans l'étage suivant du pipeline sont prises ;
      **Fin Si**
      - La classe d'instructions est retirée de l'étage courant ;
    **Fin Si**
  **Fin Si**
**Fin Pour**

Algorithme 2: Fonction qui permet d'obtenir, à partir d'un état donné, le prochain état de l'automate, avec une condition basique connue.

#### 5.3.4.4 Explosion combinatoire

Quand la complexité du modèle devient importante, l'utilisation des automates peut causer certains problèmes, comme l'explosion combinatoire du nombre d'états lors de la génération de l'automate. En effet, tel que présenté ci-dessus, le nombre d'états de l'automate est limité à $(c+1)^s$ et le nombre de transitions à $(c+2)^s \times 2^{R_{ext}}$. Donc, le nombre d'états croît exponentiellement avec la profondeur du pipeline (c'est-à-dire, en fonction de la taille du pipeline), et polynomialement en fonction du nombre de classes d'instructions. Nous pouvons distinguer trois types de processeurs :

- processeurs à architecture plus ou moins simple contenant un pipeline de faible longueur (5-6 étages), généralement utilisés dans les systèmes embarqués (4 étages sur le *C167* d'*Infineon*, par exemple). La modélisation de ce type de pipeline ne cause pas d'explosion combinatoire ;
- processeurs avec un long pipeline (avec beaucoup d'étages), appelé superpipeline (8 étages pour le *MIPS R4000*, par exemple). Ce type de processeur a, à la fois, un pipeline de profondeur élevée et de nombreuses classes d'instructions, ce qui peut conduire à l'explosion combinatoire de l'automate. Pour réduire la complexité de l'automate généré, ces superpipelines peuvent être découpés en deux pipelines (voire plus) de plus faible longueur pour générer deux (voire plusieurs) automates plus petits qui sont synchronisés en utilisant

des ressources externes [2] ;
– processeurs avec un pipeline possédant plusieurs branches, appelé superscalaire : chaque branche de pipeline peut être modélisée par un automate séparé comme présenté dans [41] (cette approche sera brièvement abordée dans la section 6.5). Les instructions sont envoyées vers les différentes branches, ainsi chaque branche a moins de classes d'instructions. Ce type de processeur peut être modélisé aussi avec plusieurs automates qu'il faut synchroniser soit en faisant le produit d'automates (pour les pipelines qui s'exécutent en parallèle) par l'intermédiaire de l'outil *a2a* fait par Vincent-Xavier Reynaud dans le cadre d'un stage de Master 2 au laboratoire IRCCyN [54], ou bien en utilisant des ressources externes (pour les pipelines qui s'exécutent séquentiellement).

## 5.4 Exemples d'application de notre approche

Deux exemples seront présentés dans cette section. Le premier est un exemple très simple qui conduit à un petit automate (l'interêt de cet exemple est que nous pouvons visualiser l'automate et ainsi vérifier le bon fonctionnement de notre approche de modélisation). Le second sera un exemple plus complexe conduisant à un automate beaucoup plus grand.

### 5.4.1 Exemple 1

Considérons un pipeline à 2 étages, nommés respectivement *Fetch* et *Execute*, avec seulement 2 instructions (*i1* et *i2*), 2 composants matériels (la mémoire (ressource externe) et une unité fonctionnelle (ressource interne)) et deux contraintes (l'accès mémoire dans l'étage *Fetch* et l'accès à l'unité fonctionnelle dans l'étage *Execute*). Supposons que l'instruction *i1* accède seulement à la mémoire dans l'étage *Fetch* alors que l'instruction *i2* accède à la mémoire dans l'étage *Fetch* et utilise l'unité fonctionnelle dans l'étage *Execute*. Nous avons donc 2 classes d'instructions :

– la classe d'instructions $A$ contenant l'instruction *i1* ;
– la classe d'instructions $B$ contenant l'instruction *i2*.

En utilisant l'outil *p2a*, cet exemple permet de générer l'automate à 9 états illustré par la figure 5.9. Chaque état de l'automate définit l'état du pipeline à un instant donné : $A$ et $B$ représentent les 2 classes d'instructions et - représente un étage vide. La condition d'une transition est formée d'une étiquette $\alpha, \beta$, où $\alpha$ représente l'instruction qui peut rentrer dans le pipeline, et $\beta$ représente l'état de

2. Le découpage d'un superpipeline en plusieurs pipelines de plus faible longueur sera étudié, plus en détail, dans le chapitre 6.

la ressource externe [3] (accès mémoire). Les notations $M$ et $/M$ signifient que la ressource externe est respectivement disponible ou occupée. La notation $X$, pour la classe d'instructions ou une ressource externe, signifie que le paramètre est sans importance pour l'état de la transition. Par exemple, l'étiquette $A, M$ signifie qu'une instruction de la classe d'instructions $A$ peut rentrer dans le pipeline et que la ressource externe $M$ est disponible.

Sur la figure 5.9, l'état initial (en tête de la figure) représente un pipeline vide. Au cycle d'horloge suivant, trois transitions peuvent être prises. Dans les conditions de ces trois transitions, une classe d'instructions ($A$ ou $B$) ou $X$ (sans importance) et l'état de la ressource externe apparaissent. Donc, une transition est prise en fonction de la classe d'instructions et de l'état de la ressource externe. Si la ressource est occupée (transition étiquetée $X,/M$), c'est-à-dire que le contrôleur d'accès mémoire n'est pas libre, aucune instruction ne peut pas entrer dans le pipeline et il restera dans le même état. Si la ressource externe est disponible, une instruction de la classe $A$ ou de la classe $B$ entre dans le pipeline pour être exécutée, et le nouvel état du pipeline sera respectivement $A$- ou $B$-.

### 5.4.2 Exemple 2

Cet exemple est plus réaliste. Il est basé sur un pipeline de 6 étages inspiré de l'architecture *DLX* pipelinée [29], en utilisant le jeu d'instructions de la *XGate* (processeur basé sur une architecture *RISC*[4]) de chez *Freescale* [19]. Les 6 étages du pipeline sont :

- un étage *Fetch* qui permet de récupérer le code de l'instruction dans la mémoire (ou le cache d'instructions) ;
- un étage *Decode* qui s'occupe du décodage de l'instruction, lit ses opérandes et exécute les instructions de branchement ;
- deux étages *Execute* qui s'occupent de l'exécution à proprement parler de l'instruction (Unité Arithmétique et Logique) ;
- un étage *Memory* qui fait un accès en mémoire (pour les instructions de chargement/rangement (*load/store*)), et un accès en lecture au banc de registres (pour les instructions de rangement (*store*)) ;
- un étage *Register* qui permet de mettre à jour le banc de registres (accès en écriture).

3. Les ressources internes sont directement résolues lorsque l'ensemble d'états suivants de l'automate est construit

4. Les processeurs *RISC* (*Reduced Instruction Set Computer*) se basent, contrairement aux processeur *CISC* (*Complex Instruction Set Computer*), sur un nombre réduit d'instructions de longueur fixe, une utilisation poussée des registres avec seulement quelques instructions permettant l'accès à la mémoire. Leur utilisation est particulièrement adaptée au pipeline.

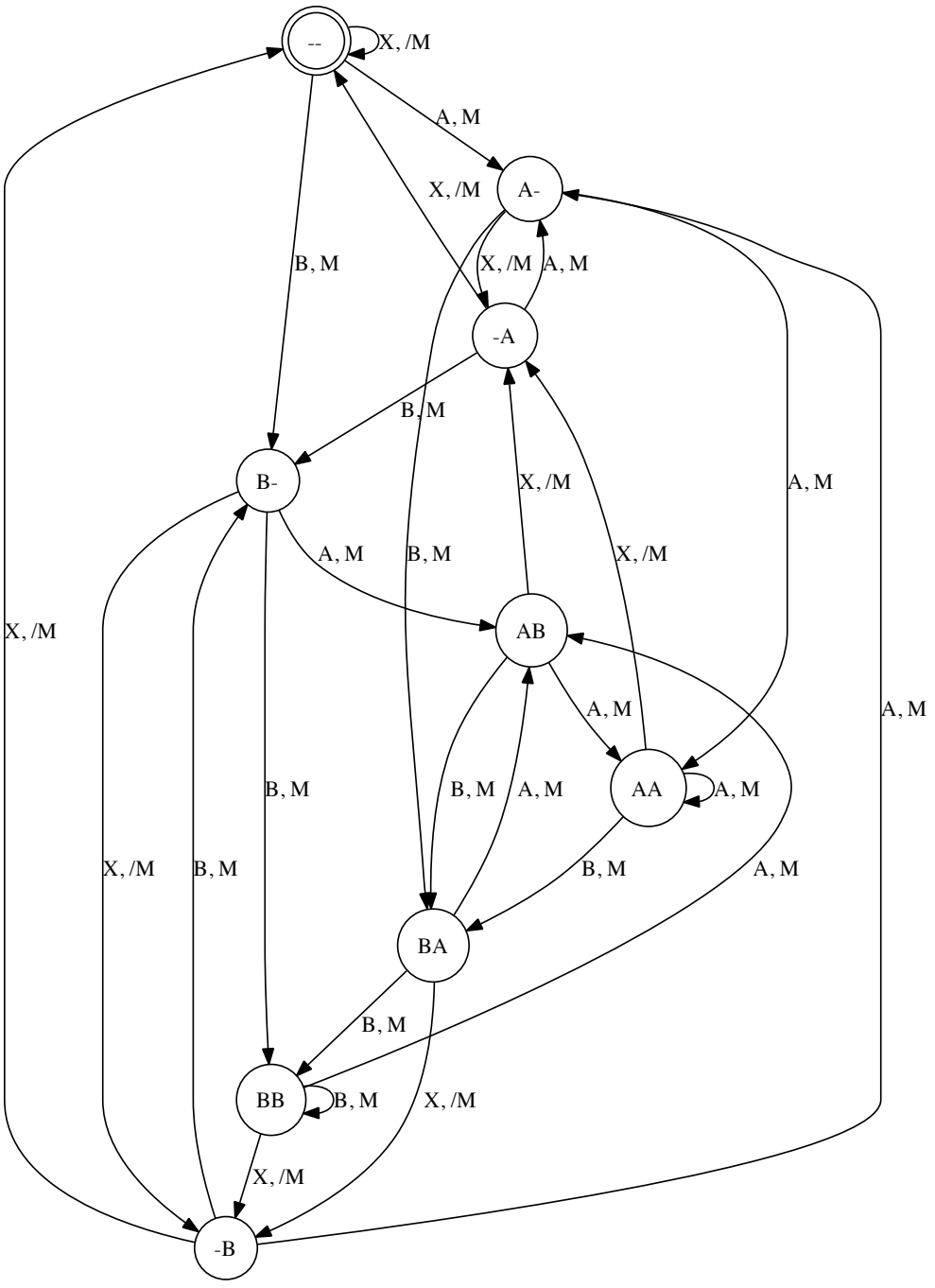



FIGURE 5.9 – Un automate à 9 états modélisant un pipeline à 2 étages.

88 instructions sont disponibles dans cet exemple. Les contraintes de concurrence sur l'utilisation des ressources matérielles sont les suivantes :

- Le banc de registres est en mesure d'effectuer, en parallèle, 3 accès en lecture et 2 accès en écriture ;
- Une architecture *Harvard* (la mémoire de données et la mémoire programme sont séparées. L'accès à chacune de ces deux mémoires s'effectue par l'intermédiaire des deux bus distincts) est utilisée ;
- Le calcul dans l'Unité Arithmétique et Logique (*UAL*) nécessite 2 étages occupés séquentiellement ;

Les classes d'instructions groupent les instructions qui utilisent les mêmes ressources (internes et externes) tels que présenté dans la section 5.3.2.

Dans cet exemple, 10 ressources sont utilisées :

- 6 ressources internes pour les étages du pipeline ;
- 1 ressource interne pour la gestion de l'Unité Arithmétique et Logique : `alu` ;
- 2 ressources externes pour l'accès mémoire : `fetch` et `loadStore` ;
- 2 ressources externes pour vérifier les dépendances de données : `dataDep1` et `dataDep2`.

Comme chacune des instructions dépend de la ressource externe `fetch` et des étages du pipeline, seulement quatre ressources (`alu`, `loadStore`, `dataDep1` et `dataDep2`) peuvent différencier les instructions : il aura un maximum de $2^4 = 16$ classes d'instructions. En se basant sur le jeu d'instructions de la *XGate*, 6 classes d'instructions sont obtenues (les autres classes correspondent à des configurations impossibles et sont supprimées).

En ce qui concerne les classes d'instructions de dépendances de données, 5 classes sont obtenues selon qu'une instruction accède ou non en lecture au banc de registres, si elle lit ses opérandes dans l'étage *Decode* ou *Memory* et si elle écrit ou non ses résultats dans les registres.

Cet exemple a été testé sur un processeur *Intel Core 2 Duo* à 2 GHz avec 2 Go de *RAM*. Les résultats sont donnés dans le tableau 5.4. Nous pouvons remarquer que pour un pipeline de 6 étages, 10 ressources (internes et externes) et 6 classes d'instructions, l'automate obtenu est assez grand (46 305 états et 307 101 transitions) mais le temps nécessaire pour générer cet automate et le code source (*C++*) correspondant ne dépasse pas les 50 s. La génération du simulateur est donc suffisamment rapide pour pouvoir modéliser des processeurs réels. Dans le cas d'un pipeline composé d'un plus grand nombre d'étages, nous pouvons le découper pour pouvoir générer des automates plus petits qui sont synchronisés pour modéliser le fonctionnement du pipeline de départ (voir la section 6.5).

| | |
|---|---|
| **Temps pour générer l'automate (s)** | 42.5 |
| **Nombre d'états de l'automate** | 46 305 |
| **Nombre de transitions dans l'automate** | 307 101 |
| **Temps pour générer le code source de l'automate (s)** | 5.2 |
| **Nombre de lignes du code source (C++)** | 85 700 |
| **Taille du code source (Mo)** | 4.7 |

TABLE 5.4 – Ce tableau présente quelques résultats sur le processus de génération du modèle interne du pipeline (automate à états finis). Les temps sont mesurés sur un processeur *Intel Core 2 Duo* à 2 GHz avec 2 Go de RAM.

## 5.5 Conclusion

Ce chapitre a présenté notre approche permettant la modélisation du fonctionnement d'un pipeline simple sous forme d'automate. Cette approche utilise une version améliorée de l'algorithme *BuildForwardFSA* pour traiter les aléas de données, les aléas de contrôle ainsi que les concurrences d'accès aux différentes ressources qui ne sont pas gérées de manière statique par l'automate. Cette amélioration se fait en utilisant des ressources externes. Deux outils *p2a* et *a2cpp* permettent de générer, à partir d'une description de pipeline et de ses aléas (structurels, de données et de contrôle), un automate et les fichiers C++ correspondants qui seront utilisés dans la simulation d'un processeur pipeliné qui sera l'objet du chapitre suivant.

# Chapitre 6

# Description de la micro-architecture

## 6.1 Introduction

Les processeurs modernes utilisent du matériel de plus en plus complexe pour extraire le parallélisme d'instructions (*ILP*) dont l'élément central est le pipeline. Dans ce chapitre, nous allons expliquer comment notre langage de description d'architecture matérielle HARMLESS permet de décrire l'architecture interne d'un processeur (la micro-architecture), dans une *vue dédiée*, afin de générer *automatiquement* un simulateur *CAS*.

Le processus de développement d'un processeur se fait en plusieurs étapes dont la finale - précédant la fabrication du processeur - est la simulation du fonctionnement de ce dernier. Les avantage de la simulation sont multiples, par exemple, nous pouvons l'utiliser pour tester une application en créant plusieurs scénarios avant le test final sur la plate-forme réelle. De plus, nous pouvons nous en servir dans le but de la conception d'une nouvelle architecture (*design Space exploration*). Une autre utilisation est de pouvoir évaluer le *WCET* pour calculer l'ordonnancement d'une application donnée [35]. Dans les trois cas, un simulateur précis au cycle près, permettant de reproduire le comportement temporel du processeur, doit être utilisé.

Pour obtenir un tel simulateur, une description à la fois fonctionnelle et temporelle du processeur est nécessaire. En d'autres termes, la génération automatique de ce type de simulateur nécessite, en plus de la description du jeu d'instructions et la description fonctionnelle des composants matériels (ALU, mémoire, registres, ...) que nous avons présentées dans le chapitre 3, la description de la micro-architecture [1], ainsi que la possibilité de gérer les différentes contraintes sur

1. Rappelons que la description du jeu d'instructions est largement *indépendante* vis-à-vis de

l'utilisation des différents composants matériels cités plus haut. En effet, il faut ajouter à la description (syntaxiquement décrite en *BNF étendu*) présentée dans la section 3.1, la description de la micro-architecture via les trois règles suivantes :

```
<pipeline> |
<machine> |
<architecture>
```

Où :
- la règle `architecture` est utilisée pour exprimer les concurrences d'accès aux composants matériels, voir section 6.2.1 ;
- la règle `pipeline` est utilisée pour décrire un pipeline, voir section 6.2.2 ;
- la règle `machine` sert à décrire l'architecture pipelinée, voir section 6.5.

Pareillement, les différentes règles peuvent être mises dans n'importe quel ordre, autant de fois que nécessaire (même 0).

La chaîne de développement autour de l'*ADL* HARMLESS pour obtenir automatiquement le simulateur *CAS* est donnée sur le schéma 6.1. L'outil *p2a* permet de générer, à partir de la description des caractéristiques du pipeline (étages, ressources matérielles utilisées dans chaque étage,...), le modèle interne du pipeline (automate à états finis, voir chapitre 5). L'outil *a2cpp* permet de générer le code *C++* correspondant (modélisant le fonctionnement du pipeline) qui, intégré dans le code *C++* du simulateur *ISS*, permet d'obtenir le simulateur *CAS*. Ces deux outils (*p2a* et *a2cpp*) pourraient être intégrés dans le compilateur, mais pour une raison de performance (optimisation de l'utilisation mémoire) nous avons préféré les séparer.

Ce chapitre présente comment les éléments de la description fonctionnelle sont mis en correspondance avec le modèle interne (le pipeline) du processeur. Au départ, nous verrons comment HARMLESS permet une description de la vue *micro-achitecture* du processeur (description d'un pipeline simple [2], ainsi que les concurrences d'accès aux composants matériels qui nous permettent de déduire les caractéristiques temporelles). Ensuite, nous allons expliquer la façon dont est fait le mappage de la vue sémantique du jeu d'instructions sur la vue micro-architecture en utilisant les accès aux composants. Puis, après avoir exposé brièvement les méthodes existantes de prédiction de branchement, nous présenterons la gestion des branchements dans notre langage. Enfin, nous expliquerons comment, dans HARMLESS, un pipeline peut être découpé en plusieurs pipelines de longueur plus faible afin de générer plusieurs automates plus petits. La description d'une architecture superscalaire sera également sommairement abordée.

---

la micro-architecture du processeur (à l'exception de quelques instructions systèmes qui modifient le cache).

2. L'expression *pipeline simple* est définie dans la section 5.1.

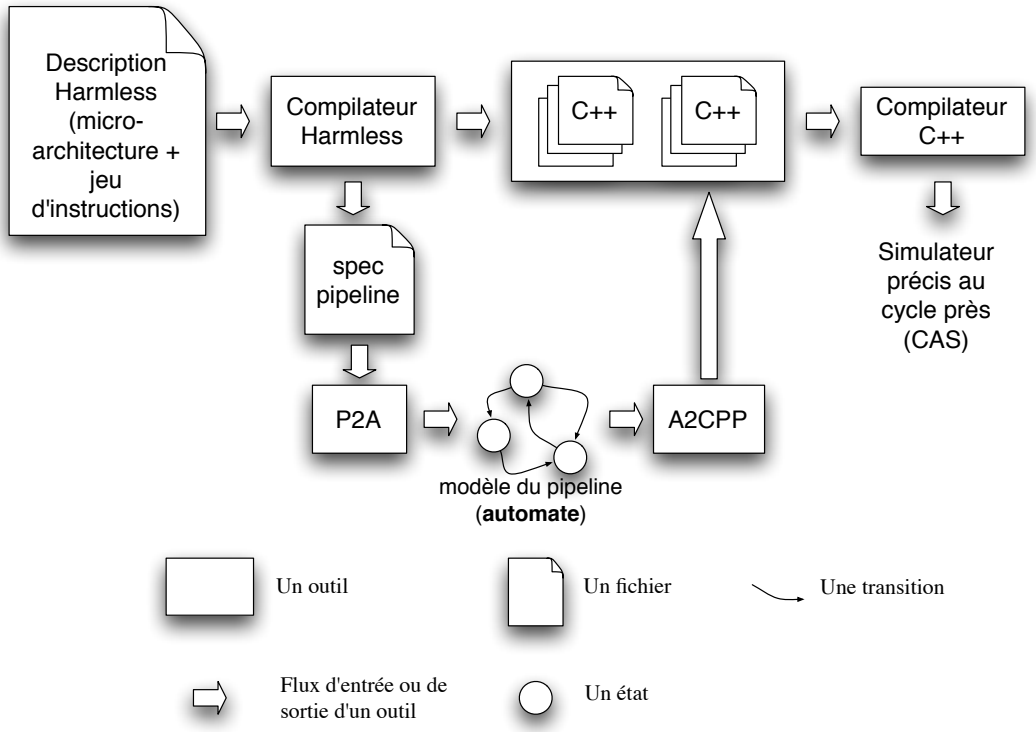



FIGURE 6.1 – Chaîne de développement du simulateur *CAS*.

## 6.2 Description de la vue micro-architecture

La vue *micro-architecture* permet de décrire l'architecture interne (pipeline et concurrences d'accès aux composants matériels) qui implémente le jeu d'instructions du processeur. Le but est de mapper la vue *sémantique* du jeu d'instructions sur la vue *micro-architecture* en utilisant les *composants* comme le montre la figure 6.2. De cette façon, un simulateur *CAS* peut être généré automatiquement. Le modèle d'exécution de ce simulateur est un automate à états finis (voir chapitre 5) généré à partir de la description des deux vues (la vue sémantique et la vue micro-architecture) du processeur dans HARMLESS.

Comme le montre la figure 6.3, la vue *micro-architecture* est décrite à travers les trois sous-vues [3] :

3. Dès le deuxième niveau de hiérarchie, nous parlons de *sous-vue* quelque soit le niveau d'imbrication.

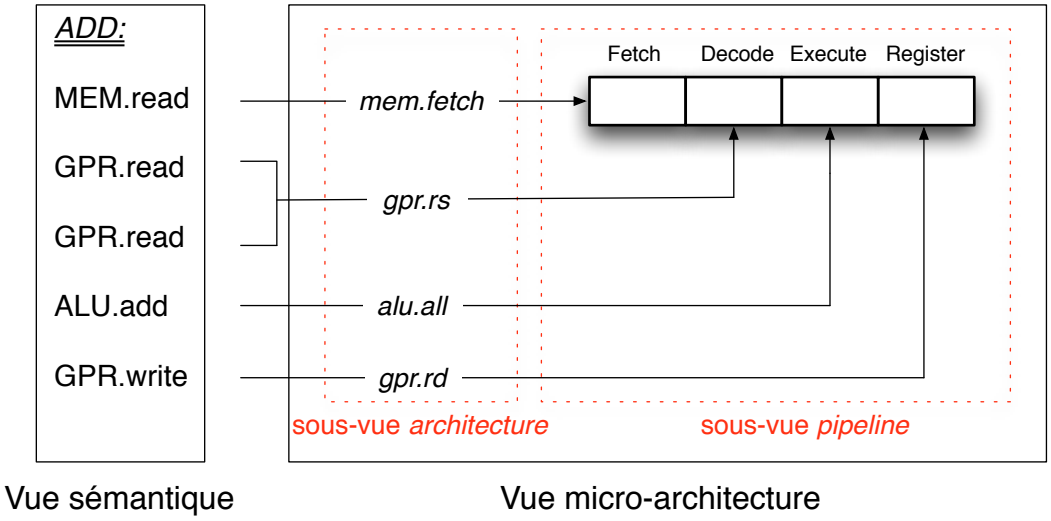



FIGURE 6.2 – Mappage de la vue sémantique du jeu d'instructions sur la vue micro-architecture en utilisant les composants. Sur la gauche, les méthodes de composants accédées par l'instruction d'addition *ADD* sont affichées. Elles sont mappées sur un pipeline de 4 étages, tout en contrôlant les concurrences d'accès aux composants décrites dans la sous-vue *architecture* (voir la section 6.2.1).

- La sous-vue `architecture` : qui permet de décrire les différentes contraintes sur l'utilisation des composants matériels (les registres, la mémoire,...) ;
- La sous-vue `machine` : qui sert à décrire l'architecture pipelinée, elle peut se composer d'un ou plusieurs pipelines (voir section 6.5 pour ce dernier cas). Son rôle est de factoriser plusieurs pipelines ; de ce fait, elle aurait pu être omise lorsque nous avons un seul pipeline ;
- La sous-vue `pipeline` : qui permet de décrire un pipeline.

Il est important de noter que dans HARMLESS, la description de la micro-architecture est très concise (quelques dizaines de lignes). Cette propriété permet de réduire le temps nécessaire à la description. De plus, le découplage entre la description fonctionnelle d'un processeur et celle de la micro-architecture permet de générer, séparément et simultanément, les deux simulateurs : *ISS* et *CAS*. Par conséquent, il est tout à fait envisageable, à partir d'un unique jeu d'instructions, de proposer plusieurs architectures internes en modifiant, par exemple, la profondeur du pipeline ou les concurrences d'accès aux différents composants dans le but d'exploration d'une nouvelle architecture (*design space exploration*).

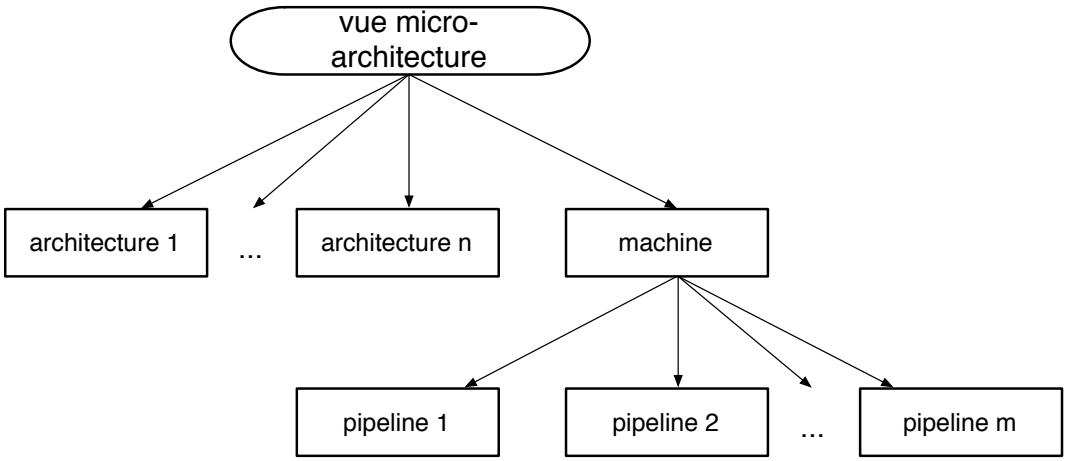



FIGURE 6.3 – La vue micro-architecture est composée de plusieurs sous-vues : une ou plusieurs sous-vues `architecture` et une sous-vue `machine`. Cette dernière peut elle-même être raffinée en une ou plusieurs sous-vues de description du/des pipelines de la micro-architecture.

### 6.2.1 La sous-vue `architecture`

Comme le montre la figure 6.2, la sous-vue `architecture` constitue l'interface entre l'ensemble des composants matériels (les registres, la mémoire, ALU,...), défini dans la section 3.3, et la définition du pipeline (section 6.2.2). En effet, à travers les étages du pipeline, les instructions accèdent aux différents composants matériels, cet accès est contrôlé par la sous-vue `architecture`. Donc, cette sous-vue permet d'exprimer les contraintes matérielles ayant des conséquences sur la séquence temporelle du simulateur. Par exemple, dans cette sous-vue, nous pouvons définir que les registres sont en mesure d'effectuer, en parallèle, 2 accès en lecture.

Nous pouvons trouver, dans une description, plusieurs sous-vues `architecture` (voir figure 6.3). Ceci est utile pour pouvoir montrer différents scénarios possibles d'accès aux composants matériels. La structure générale de cette sous-vue est de la forme :

```
1 architecture <name>{
2    device <device_name> : <component_name> {
3       <device_body>
4    }
5    ... -- autres 'devices'
```

```
6 }
```

Elle contient de nombreux dispositifs (`device`) afin de contrôler la concurrence entre les instructions pour accéder au même composant (`component`) décrit ailleurs dans la description. Chaque dispositif (`device`) est relié à un seul composant matériel dont le nom est précisé dans le champ `<component_name>`. La notion du `device` n'est pas pleinement exploitée à l'heure actuelle. Nous l'avons conçue pour éventuellement permettre une instantiation d'un même composant générique dans la description d'une architecture (par exemple, plusieurs bancs de registres).

La partie `<device_body>` est une succession d'éléments tels que :

- définition des *alias* qui permettent de remplacer plusieurs méthodes d'un composant donné et qui sont régulièrement utilisées. Ils sont aussi réutilisables dans la sous-vue `pipeline` (voir la section 6.2.2). Par exemple, un alias `read` peut être défini pour désigner *read8* ou *read16* selon qu'une instruction accède à un composant (comme la mémoire) pour lire 8 ou 16 bits respectivement de la manière suivante :

```
1 read  is read8  or read16
```

  Le mot clé `or` est équivalent à un *ou exclusif*;
- `port` qui permet de contrôler la concurrence au cours de l'accès à une ou plusieurs méthodes d'un composant. Un port peut être *privé* à la micro-architecture, ce type de ports représente alors les *ressources internes*, ou bien il peut être *partagé* (c'est-à-dire que le port n'est pas utilisé exclusivement par le pipeline), dans ce cas il représente les *ressources externes*. Par exemple, si le banc de registres d'un processeur donné est en mesure d'effectuer, en parallèle, 3 accès en lecture et 2 accès en écriture, nous définissons deux ports pour contrôler ces accès de la manière suivante :

```
1 port rs : read (3);
2 port rd : write (2);
```

  Ces deux ports (`rs` et `rd`) sont privés vis à vis du pipeline. Pour qu'un port soit déclaré comme partagé, il faut faire précéder sa description par l'opérateur `shared`.

Avant de passer à la sous-vue `pipeline`, nous allons expliquer comment la mémoire influe sur le comportement temporel d'un processeur.

**Hiérarchie mémoire** La hiérarchie mémoire est un mécanisme qui influe aussi dans une large mesure sur le comportement temporel d'un processeur. Cependant, la description de la hiérarchie mémoire paraît plus simple que l'architecture interne du processeur :

- la hiérarchie mémoire n'influe sur le reste du système que par le blocage sur un accès mémoire, ce qui est déjà pris en compte dans la description de la micro-architecture à travers la section `architecture` (plus précisément, à travers les ressources externes) ;
- il existe des politiques de cache classiques (LRU, Random, ..) qui seront certainement directement intégrées dans le compilateur, sous la forme de squelette par exemple.

Pour le moment, HARMLESS ne permet pas de décrire les différents niveaux de mémoire constituant la hiérarchie mémoire. Les éléments mémoires sont simplements décrits dans un `component` (taille, largeur de bus,...). (Voir section 3.3).

### 6.2.2 La sous-vue `pipeline`

La sous-vue `pipeline` permet de décrire un pipeline mappé sur une `architecture`. La syntaxe générale de description d'un pipeline est la suivante :

```
1 pipeline <name> maps to <architecture_name> {
2    stage <stage_name> {
3       <stage_body>
4    }
5    ... -- autres etages
6 }
```

Dans la description d'un pipeline, tous les étages sont énumérés dans l'ordre. Pour chaque étage, les composants (mémoire, registres,...) et les méthodes (lecture, écriture,...), qui peuvent être accédés par le jeu d'instructions, sont énumérés dans la partie `<stage_body>` comme le montre l'exemple suivant :

```
1 stage Memory {
2       MEM : read, write; -- acces en lecture et en ecriture au
            composant MEM
3       GPR : read; -- acces en lecture au composant GPR
4 }
```

Les composants et les méthodes associées seront mappés sur les dispositifs et les ports de la sous-vue `architecture` durant la phase compilation, permettant ainsi de découper le comportement des instructions en plusieurs étapes correspondant aux différents étages du pipeline (voir section 6.3).

De plus, dans HARMLESS, les raccourcis matériels (appelés également court-circuits ou envois) peuvent être modélisés. Cette technique consiste à renvoyer le résultat d'une opération aux instructions le nécessitant pour pouvoir avancer dans le pipeline, et ceci avant qu'il ne soit disponible dans le registre destination. Elle permet de diminuer les suspensions du pipeline dues aux aléas de données (*LAE*).(Un exemple est donné dans la section 6.3).

À partir de ces trois sous-vues (`architecture`, `pipeline` et `machine`) ainsi que la vue sémantique décrivant le comportement du jeu d'instructions (voir section 3.2.1.5), un fichier [4], contenant des informations sur le pipeline, les différentes ressources internes et externes, la définition des classes d'instructions (voir section 5.3.2) et des classes d'instructions de dépendance de données (voir section 5.3.3), est généré. Ensuite, les deux outils *p2a* et *a2cpp* permettent de générer respectivement le modèle du pipeline (l'automate) et le code *C++* correspondant qui, ajouté au code *C++* du simulateur *ISS*, nous permettra d'obtenir un simulateur précis au cycle près. Les différentes étapes de génération de ce type de simulateur seront expliquées en détail dans le chapitre 7.

## 6.3 Exemple de description d'une micro-architecture

Reprenons le même exemple que dans la section 5.4.2, mais en ajoutant un court-circuit permettant d'envoyer le résultat, avant qu'il ne soit disponible dans le registre destination, à l'étage du pipeline qui s'occupe de l'exécution des instructions. Rappelons que les contraintes de concurrence sur l'utilisation des ressources matérielles sont les suivantes :

- Les registres sont en mesure d'effectuer, en parallèle, 3 accès en lecture et 2 accès en écriture ;
- Une architecture *Harvard* est utilisée ;
- Le calcul dans l'unité arithmétique et logique nécessite 2 étages et n'est pas pipeliné.

Comme nous l'avons déjà expliqué, dans HARMLESS, la description de la micro-architecture se fait au travers de trois sous-vues : `architecture`, `pipeline` et `machine`. La première sous-vue `architecture` est définie comme suit :

4. Un exemple de ce fichier est donné dans l'annexe C.

```
1 architecture Generic {
2   device gpr : GPR {
3       read  is read8  or read16
4       write is write8 or write16
5       port rs : read  (3);  -- 3 lectures en parallele
6       port rd : write (2);
7     }
8   device alu : ALU {
9       port all;
10    }
11  device mem : MEM {
12      read  is read8  or read16
13      write is write8 or write16
14      shared port fetch : read; -- une lecture
15      shared port loadStore : read or write; --une lecture ou
            une ecriture
16    }
17  device fetcher : Fetcher {
18      port branch : branch;
19    }
20 }
```

Dans cette description, une `architecture`, nommée `Generic` et contenant plusieurs dispositifs (`device`), est déclarée. Par exemple, le dispositif `mem` contrôle les concurrences d'accès au composant `MEM` (la mémoire) par deux ports partagés `fetch` et `loadStore`. Le port `loadStore` permet l'accès aux deux méthodes `read` et `write`. L'opérateur `or` est équivalent à un *ou exclusif*, et ces deux méthodes sont des alias pour des méthodes décrites dans la section `component` (`read8 or read16` pour `read` et `write8 or write16` pour `write`). De cette façon, à un moment donné, si une instruction utilise `read` dans un étage de pipeline, la deuxième méthode `write` devient inaccessible, et la ressource associée est définie comme occupée.

Parfois, l'utilisation de n'importe quelle méthode d'un composant, par une instruction en exécution dans un étage du pipeline, le rend indisponible aux autres instructions. Au lieu de forcer l'utilisateur à donner la liste de toutes les méthodes du composant en question, une liste vide est interprétée comme une liste intégrale des méthodes. Ici, le port `alu` utilise cette propriété.

Passons maintenant à la sous-vue `pipeline`. Elle peut être décrite comme l'illustre l'exemple ci-dessous.

```
1  pipeline sample_pipe maps to Generic {
2    stage Fetch {
3        MEM : read;
4      }
5    stage Decode {
6        Fetcher : branch;
7        GPR : read;
8      }
9    stage Execute1 {
10       ALU release in <Execute2> : *; -- '*' signifie que toute
             methode de l'ALU peut etre utilisee
11     }
12   stage Execute2 {
13     }
14   stage Memory {
15       MEM : read, write;
16       GPR : read;
17     }
18   stage Register {
19       GPR bypass in <Execute2> : write;
20     }
21 }
```

Dans cette description un pipeline à 6 étages, nommé `sample_pipe`, est declaré et mappé sur l'architecture `Generic` décrite ci-dessus. Dans cette sous-vue, les composants et leurs méthodes ainsi que les deux alias (`read` et `write`) sont utilisés. Les différents étages du pipeline sont énumérés :

- *Fetch* qui permet de récupérer le code de l'instruction dans la mémoire ;
- *Decode* qui s'occupe du décodage de l'instruction, lit ses opérandes et exécute les instructions de branchement ;
- *Execute1* et *Execute2* qui s'occupent de l'exécution de l'instruction ;
- *Memory* qui fait un accès en mémoire, et un accès en lecture au banc de registres ;
- *Register* qui permet d'écrire le résultat dans un registre. Un court-circuit permet d'envoyer le résultat à l'étage *Execute2* du pipeline.

Lorsqu'un port est utilisé dans un étage de pipeline, il est implicitement pris au début de l'étage et relâché à la fin de cet étage. Si un port doit être détenu pendant plus d'un étage, l'étage dans lequel il sera libéré, est explicitement donné. Ici, le port `all` de `alu` est pris dans l'étage `Execute1` et libéré dans l'étage `Execute2`.

Dans cette sous-vue, les composants et leurs méthodes (définis dans la section 3.3) seront mappés sur les dispositifs et les ports de l'architecture `Generic` comme

le montre la figure 6.4. À ce niveau, il est possible de vérifier que chaque instruction peut s'exécuter dans le pipeline, à condition que l'ordre d'accès aux méthodes des composants matériels défini dans le pipeline corresponde à l'ordre d'accès défini dans la sémantique de l'instruction.

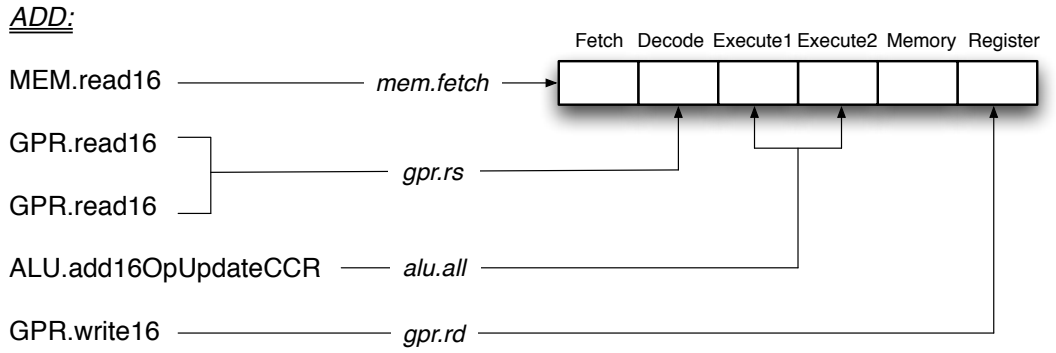



FIGURE 6.4 – Mappage de la vue sémantique du jeu d'instructions sur la vue micro-architecture en utilisant les accès aux composants. Sur la gauche, les méthodes de composants accédées par l'instruction d'addition *ADD* sont affichées. Elles sont mappées sur un pipeline de 6 étages, en utilisant les dispositifs (`mem`, `gpr` et `alu`) et les ports (`fetch`, `rs`, `rd` et `all`) qui contrôlent les concurrences d'accès aux composants.

Dans cet exemple, l'architecture pipelinée se compose d'un seul pipeline (`sample_pipe`). La sous-vue `machine` est donc définie comme suit :

```
1 machine all {
2   sample_pipe
3 }
```

Dans cette description, une sous-vue `machine` nommée `all` permet de construire le diagramme de pipeline du processeur.

À partir de ces trois sous-vues (`architecture`, `pipeline` et `machine`) et en se basant sur la vue sémantique décrivant le comportement des instructions du co-processeur *RISC*, le *XGate*, du *HCS12X* de chez *FreeScale* [19], le fichier presenté dans l'annexe C est généré [5]. Ensuite, en utilisant les deux outils *p2a* et *a2cpp*, le code *C++*, permettant de simuler le fonctionnement d'un pipeline, sera obtenu.

5. Les différentes étapes de génération seront expliquées en détails dans le chapitre 7.

Enfin, ce code, ajouté au code du simulateur de jeu d'instructions du *XGATE*, permet d'obtenir un simulateur précis au cycle près de ce processeur (voir la figure 6.1).

## 6.4 Gestion des branchements dans HARMLESS

Avec l'apparition des processeurs pipelinés, les instructions de branchement ont commencé à avoir un grand impact sur les performances. En effet, la direction prise par un branchement est calculée, dans la plupart des cas, assez tard dans le pipeline, ce qui implique que l'exécution des instructions qui suivent l'instruction de branchement a déjà été commencée. Dans le cas où le branchement sera pris (c'est-à-dire qu'il va changer la valeur du compteur programme), ces instructions ne doivent pas être exécutées.

Dans HARMLESS, pour gérer les branchements dans un pipeline, la méthode utilisée consiste à insérer des bulles derrière l'instruction de branchement tant que la direction du branchement (pris ou non pris) n'est pas connue, ce qui évite d'avoir à commencer des instructions qu'il faudra annuler par la suite si le branchement est résolu comme pris. D'autres techniques [29], plus efficaces, comme le branchement différé, le déroulage des boucles,..., ne sont pas encore implémentées et seront ajoutées dans les travaux futurs.

### 6.4.1 Principes de la prédiction de branchement

Afin de réduire les pénalités de branchements dans un pipeline, il existe des méthodes de prédiction qui sont classées en deux catégories :

- statiques : prédiction faite à la compilation ;
- dynamiques : prédiction dépendant du comportement dynamique du branchement pendant l'exécution d'un programme.

Avant de détailler ces deux méthodes de prédiction, nous allons parler de la mémoire cache permettant de mémoriser l'adresse destination d'un branchement pris.

#### 6.4.1.1 La mémoire cache d'adresse destination (*BTB*)

Dans le cas d'un branchement prédit pris, soit par une méthode statique soit par une méthode dynamique, il faut savoir l'adresse de l'instruction cible de branchement pour la charger avant même qu'elle ne soit calculée.

Pour résoudre ce problème, une technique, basée sur une mémoire cache d'adresse destination connue sous le nom de *BTB* (pour *Branch Target Buffer*)

[34], a été utilisée. En effet, un branchement, déjà rencontré dans un programme donné, saute généralement à la même adresse cible. Donc, s'il y a moyen de se souvenir où il a été la dernière fois (en stockant l'adresse cible dans la *BTB*), cette adresse peuvent être utilisée avant de connaître le résultat du branchement. Bien entendu, la taille de la *BTB* est limitée et elle ne permet que de stocker les derniers branchements rencontrés (identifiés par leur adresse dans le programme à exécuter).

#### 6.4.1.2 Méthodes statiques

Les méthodes statiques consistent à prédire le comportement du branchement via des informations connues à la compilation : direction du branchement ou autres informations. La méthode statique, la plus simple, consiste à examiner la structure du programme et utiliser des informations collectées sur des exécutions précédentes du programme et prédire que le branchement est toujours non pris ou bien toujours pris, cette technique est connue sous le nom *profilage*.

Des améliorations ont été apportées sur cette méthode pour diminuer la pénalité due aux branchements mal prédis, comme par exemple, prédire les branchements arrières (branchements des boucles) comme pris et les autres comme non pris. Par exemple, le microprocesseur *PowerPC 601* [5] utilise cette méthode afin de prédire la direction des branchements. Dans ce processeur, l'algorithme de prédiction prédit que le branchement sera pris si le placement de l'adresse cible est négative et prédit qu'il sera non pris si l'adresse est positive. Pour permettre au compilateur de décider la direction de prédiction, un bit dans le code-opération de l'instruction de branchement permet d'inverser le schéma de prédiction. c'est-à-dire, si une instruction de branchement arrière n'est pas une instruction de boucle et que le compilateur estime qu'elle doit être prédite non prise, il positionne le bit et cela est pareil pour une instruction de branchement avant qu'il veut prédire comme prise.

#### 6.4.1.3 Méthodes dynamiques

Quand la profondeur du pipeline est faible, les techniques statiques peuvent être utilisées et la pénalité due à un branchement mal prédit reste faible. Mais avec l'apparition des superpipelines et des pipelines superscalaires, ces méthodes s'avèrent insuffisantes, d'où la nécessité des méthodes de prédiction dynamique des branchements afin de pouvoir charger le pipeline avec le bon flux d'instructions.

La méthode dynamique la plus simple nécessite une *BTB* et un prédicteur, utilisant un compteur 1 bit à saturation, qui se souvient de la direction prise par chaque branchement (pris ou non pris). Si le branchement a été pris, le compteur s'incrémente (bit =1), et le prochain branchement sera prédit pris, et l'exécution

du pipeline continue à partir de l'adresse destination stockée dans la *BTB*. Après l'évaluation de la condition, si le branchement est bien prédit, le compteur garde sa valeur (bit=1), sinon le compteur se décrémente (bit=0), et ainsi de suite. Cette méthode a été améliorée en utilisant un prédicteur avec un compteur à 2 bits [56] (prédicteur local), donnant ainsi des résultats bien plus fiables surtout dans le cas des boucles imbriquées. Dans ce cas, chaque entrée dans la *BTB* est associée à un compteur à 2 bits ; elle stocke à la fois l'adresse de l'instruction du branchement, l'adresse de l'instruction cible et la prédiction sous forme de 2 bits (00 : non pris, 01 : faiblement non pris, 10 : faiblement pris, 11 : pris). Le principe de fonctionnement de ce prédicteur de branchement est très proche de celui à 1 bit et est donné sur la figure 6.5.

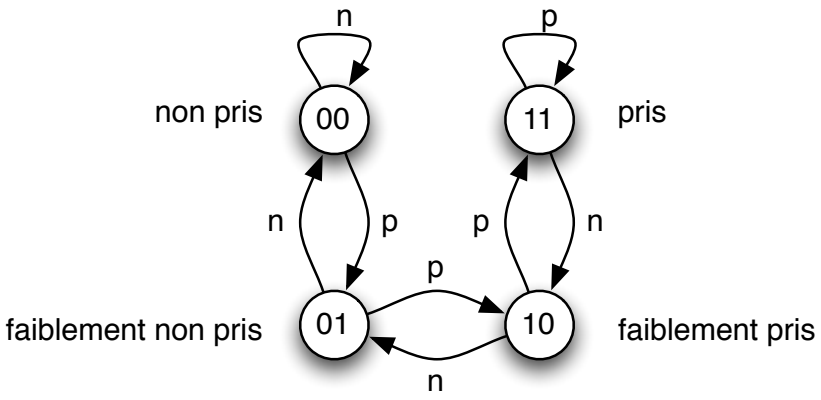



FIGURE 6.5 – Fonctionnement d'un compteur 2 bits. p : branchement pris et n : branchement non pris. Supposons que le compteur d'une instruction de branchement soit dans l'état 00, ce branchement sera prédit non pris. Après l'évolution de la condition, si le branchement est bien prédit, le compteur garde sa valeur 00. Sinon le compteur s'incrémente et un nouvel état 01 est obtenu.

D'autres méthodes plus complexes, mais en même temps beaucoup plus fiables, existent comme le prédicteur global à 2 niveaux [60], G-share et exécution spéculative, le prédicteur hybride (certains branchements sont mieux prédits par un prédicteur local et d'autres par un prédicteur global) ([29], [38])... Ces prédicteurs ne sont pas encore abordés dans HARMLESS.

### 6.4.2 HARMLESS et la prédiction de branchement

Pour le moment, dans HARMLESS, la prédiction dynamique de branchement, utilisant le compteur à 1 ou 2 bits, peut être décrite et permet de générer un

prédicteur de branchement dans le simulateur *CAS*. Les autres méthodes, permettant la réduction encore plus efficace des pénalités dues aux branchements mal prédits, seront ajoutées par la suite.

La description de la *BTB* ainsi que celle de la politique de prédiction de branchements sont associées directement au composant chargé de la gestion du compteur programme.

Prenons par exemple, le cœur *e200z1* qui est présent dans le *PowerPC5516* de chez *Freescale* (voir [21]). L'unité de branchement de ce cœur contient une *BTB* à 4 entrées, pouvant stocker les 4 derniers branchements rencontrés dans le programme. Le *e200z1* utilise un mécanisme de prédiction dynamique de branchement avec un compteur à 2 bits. Chaque entrée de la *BTB* contient :

- un champ pour stocker l'adresse de l'instruction de branchement ;
- un champ pour mémoriser l'adresse de l'instruction cible ;
- un champ contenant la prédiction sous forme de 2 bits (00 : non pris, 01 : faiblement non pris, 10 : faiblement pris, 11 : pris).

La valeur à laquelle le prédicteur est initialisé la première fois est égale à 10 (faiblement pris). Les entrées dans la *BTB* sont attribués pour les instructions de branchement prises en utilisant un algorithme de remplacement *FIFO*.

La syntaxe générale de la définition d'une *BTB* dans HARMLESS est donnée par :

```
1  BTB <btb_name> {
2    size := <btb_size>
3    counter := <1_or_2>
4  }
```

Dans HARMLESS, 4 méthodes (au niveau description) sont prédéfinies pour gérér la *BTB* :

- `u1 isInBtb (< pc_type> < pc_name>)` : cette fonction retourne 1, si l'instruction de branchement qui se situe à l'adresse `<pc_name>` dans le programme à exécuter existe dans la *BTB*, sinon elle retourne 0 ;
- `u1 getPrediction ()` : cette fonction retourne 1 si le branchement est prédit pris sinon elle retourne 0 ;
- `void decrementCounter ()` : tant que la valeur du compteur n'est pas saturée (c'est-à-dire, égale à 0 ou 00), cette fonction la décrémente si le branchement n'était pas pris ;
- `void incrementCounter ()` : tant que la valeur du compteur n'est pas saturée (c'est-à-dire, égale à 1 ou 11), cette fonction l'incrémente si le branchement était pris ;
- `void storeInBtb (<pc_type> <pc_name>)` : cette fonction ajoute l'instruc-

tion de branchement ayant comme adresse `<pc_name>` à la *BTB* en utilisant un algorithme de remplacement *FIFO*.

La *BTB* du cœur *e200z1* ainsi que le mécanisme de prédiction dynamique de branchement sont, actuellement, décrits dans le composant `fetcher` (chargé de la gestion du compteur programme) au lieu d'être inclus dans la vue *micro-architecture*, de la façon suivante :

```
1  component fetcher {
2  -- declaration du compteur programme
3    program counter u32 PC;
4
5  -- initialisation du compteur programme
6     void reset() {
7       PC := 0;
8    }
9
10   BTB btb5516 {
11     size := 4
12     counter := 2
13   }
14
15 -- "newPC" represente l'adresse cible de branchement
16 -- "condition" represente la condition de branchement
17 -- si le branchement est inconditionnel, "condition"=1
18  void absBranch(u32 newPC, u1 condition)
19  {
20   u1 isInBtb := fetcher.btb5516.isInBtb(PC)
21   if (isInBtb) then
22     u1 prediction := fetcher.btb5516.getPrediction()
23     if prediction != condition then
24       stall 1 cycle -- ajout d'une bulle apres l'instruction
             branchement
25     end if
26     if (! condition) then
27       fetcher.btb5516.decrementCounter()
28     else
29       fetcher.btb5516.incrementCounter()
30     end if
31   end if
32   if (cond) then
33      if (!isInBtb) then
34        stall 1 cycle -- ajout d'une bulle apres l'instruction
```

```
            branchement
35          fetcher.btb5516.storeInBtb(PC)
36        end if
37        PC := newPC -- "newPC" represente l'adresse cible de
              branchement
38      end if
39    }
```

## 6.5 Possibilité de découper un pipeline

HARMLESS offre la possibilité de couper un pipeline (ou un superpipeline) en plusieurs pipelines de plus faible profondeur pour pouvoir générer des automates plus petits, permettant ainsi de résoudre le problème d'explosion combinatoire (voir section 5.3.4.4).

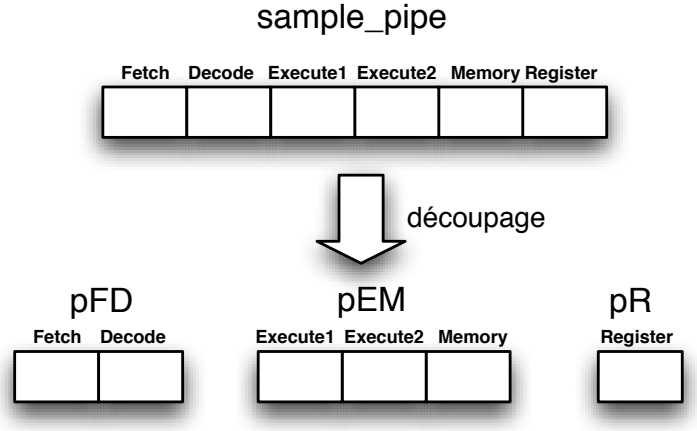



FIGURE 6.6 – Exemple de découpage d'un pipeline pour pouvoir générer plusieurs petits automates à états finis (modèle interne du pipeline).

L'idée principale consiste à décrire chaque pipeline séparément et ensuite faire l'assemblage du tout. Reprenons par exemple, le pipeline de 6 étages (`Fetch`, `Decode`, `Execute1`, `Execute2`, `Memory` et `Register`) décrit dans la section 5.4.2. Essayons maintenant de le découper en 3 pipelines comme le montre la figure 6.6.

La description dans HARMLESS se fait de la manière suivante :

```
1 pipeline pFD maps to Generic {
2   stage Fetch {
3       MEM : read;
4     }
5   stage Decode {
6       Fetcher : branch;
7       GPR : read;
8     }
9 }
10
11 pipeline pEM maps to Generic {
12   stage Execute1 {
13       ALU release in <Execute2> : *; -- '*' signifie que toute
             methode de l'ALU peut etre utilisee
14     }
15   stage Execute2 {
16     }
17   stage Memory {
18       MEM : read, write;
19       GPR : read;
20     }
21 }
22
23 pipeline pR maps to Generic {
24   stage Register {
25       GPR : write;
26     }
27 }
```

Dans cette description, les trois pipelines (`pFD`, `pEM` et `pR`) sont mappés sur une architecture nommée `Generic` décrite ailleurs (par exemple, voir 6.3).

Dans HARMLESS, pour reconstruire le pipeline d'origine, il suffit de définir l'ordre dans lequel les 3 pipelines s'exécutent par la sous-vue `machine` comme suit :

```
1 machine all {
2   pFD, pEM, pR
3 }
```

Dans cette description, une sous-vue `machine` nommée `all` permet de construire le diagramme de pipeline du processeur. L'opérateur « `,` » signifie que

les 3 pipelines sont séquentiels.

La synchronisation entre les différents pipelines sera expliquée en détails dans le chapitre suivant.

Cette approche permet une ouverture vers les architectures superscalaires.

## Extension au superscalaire

Un processeur superscalaire est un processeur capable d'exécuter plusieurs instructions simultanément. C'est une architecture à plusieurs pipelines s'exécutant en parallèle, elle contient plusieurs unités de calcul.

L'idée principale consiste à décrire chaque pipeline de l'architecture supescalaire séparément et ensuite faire l'assemblage du tout. Considérons, par exemple, le processeur *PowerPC 750* de *Freescale* [20]. C'est un processeur *RISC* 32 bits superscalaire de degré 2, donc il est capable de compléter l'exécution des deux instructions simultanément. Il intègre six unités d'exécution :

- Unité de calcul en virgule flottante (*FPU* pour *Floating-Point Unit*) : destinée à effectuer les opérations en virgule flottante (comme addition, multiplication,...). La *FPU* est pipelinée, les tâches qu'elle effectue sont divisées en sous-tâches, mises en œuvre par trois étages successives ;
- Unité de traitement des branchements (*BPU* pour *Branch Processing Unit*) : qui s'occupe des instructions de branchement ;
- Unité contrôlant les registres du système (*SRU* pour *System Register Unit*) ;
- Unité de chargement/rangement (*LSU* pour *Load/Store Unit*) : qui s'occupe de toutes les instructions de chargement et de rangement. En d'autres termes, les instructions qui accèdent à la mémoire. La *LSU* est pipelinée, elle est formée de 2 étages ;
- Deux unités entières *IU1* et *IU2* (*IU* pour *Integer Unit*) : *IU1* peut exécuter toutes les instructions entières. Alors que, IU2 permet d'exécuter toutes les instructions entières à l'exception des instructions de multiplication et de division.

Comme ce microprocesseur possède 6 unités d'exécution indépendantes, donc 6 instructions peuvent être exécutées simultanément. Par contre, au plus deux instructions peuvent être décodées en parallèle. D'où, le débit d'instructions global est égal à 2 instructions par cycle d'horloge. Le pipeline principale du *PowerPC 750* est formé de 4 étages : *Fetch*, *Decode*, *Execute* et *WriteBack*. L'étage *Execute* est décomposé en 5 sous-pipelines (un pipeline par unité d'exécution à l'exception de la *BPU* qui va être décrite, dans HARMLESS, par la *BTB* et la politique de prédiction de branchement). Le diagramme de pipeline du *PowerPC 750* est donné sur la figure 6.7.

Dans HARMLESS, Les pipelines $p1$ à $p7$ seront décrits comme un pipeline simple chacun et l'assemblage du tout sera fait par la sous-vue `machine` comme

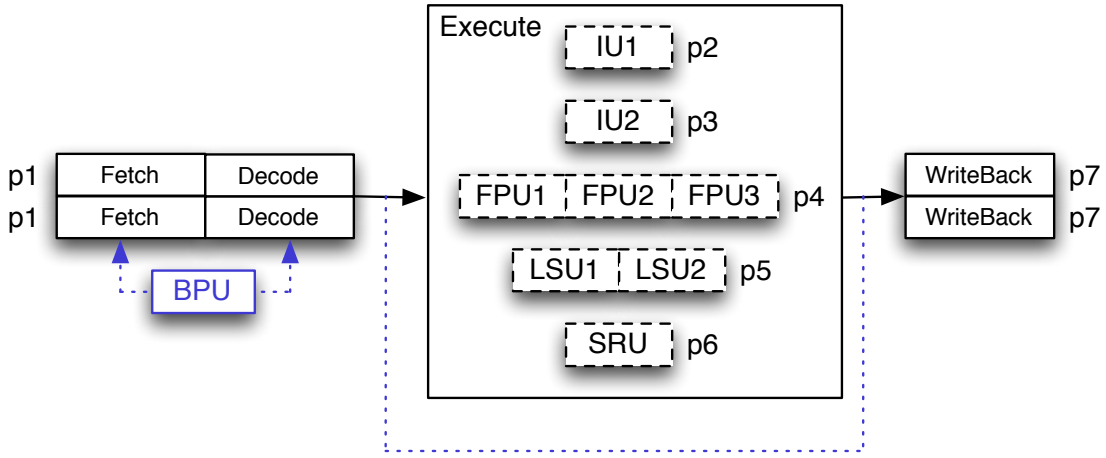



FIGURE 6.7 – Architecture superscalaire du *PowerPC 750*. Il possède 6 unités d'exécution qui peuvent être utilisées en parallèle.

suit :

```
machine ppc750 {
  p1 fetch1 | p1 fetch2, (p2|p3|p4|p5|p6), p7 finish1 | p7
     finish2
}
```

Dans cette description, une sous-vue `machine` nommée *ppc750* permet de construire le diagramme de pipeline superscalaire du processeur. L'opérateur « | » signifie que les pipelines s'exécutent en parallèle, et l'opérateur « , » signifie que les pipelines sont séquentiels. L'opérateur « | » a une priorité plus élevée que l'opérateur « , ». *fetch1*, *fetch2*, *finish1* et *finish2* sont des étiquettes qui permettent de décrire le fait que deux instructions peuvent être décodées simultanément et deux instructions peuvent écrire leur résultat simultanément.

## 6.6 Conclusion

Ce chapitre a présenté la manière dont HARMLESS permet la description de la micro-architecture d'un processeur pipeliné à travers les trois sous-vues : `architecture`, `pipeline` et `machine`. De plus, il a présenté comment un pipeline peut être découpé en plusieurs pipelines de plus faible profondeur pour générer

plusieurs petits automates.

La description de la micro-architecture, ajoutée à la description du jeu d'instructions, permet de générer un simulateur précis au cycle près (*CAS*). Il est important de signaler que la description du jeu d'instructions est indépendante de la description de la micro-architecture (ce qui n'est pas forcément vrai comparé aux autres *ADL*, où l'utilisateur est obligé de réécrire le comportement de toutes les instructions en le spécifiant par étage du pipeline). Par conséquent, dans HARMLESS, la description de la micro-architecture est très concise (quelques dizaines de lignes) et peut se faire en très peu de temps.

Dans le chapitre suivant, nous allons parler des différentes étapes permettant la génération automatique du simulateur *CAS* à partir de la description. De plus, quelques résultats seront donnés et analysés.

# Chapitre 7

# Génération du simulateur *CAS*

## 7.1 Introduction

Dans ce chapitre, nous aborderons les différentes étapes permettant la génération du simulateur *CAS* à partir de la description fonctionnelle du jeu d'instructions (chapitre 3) et la description de la micro-architecture du processeur (chapitre 6).

Pour générer ce type de simulateur, nous allons faire le lien entre la description de la micro-architecture d'une part et le modèle interne du pipeline (chapitre 5) d'autre part. Nous repartons de la figure 6.1 représentant la chaîne de développement associée à l'*ADL* HARMLESS pour obtenir le simulateur *CAS*. Dans ce chapitre notre intérêt porte sur la partie entourée d'un cercle pointillé sur la figure 7.1, c'est-à-dire sur la partie du compilateur s'occupant de la génération, à partir de la description de la vue micro-architecture, du fichier décrivant le pipeline à modéliser [1] (les étages, les classes d'instructions, les classes d'instructions de dépendance de données et les ressources internes et externes). L'outil *p2a* accepte en entrée ce fichier et le traite. En sortie, nous obtenons le modèle interne du pipeline (automate à états finis) qui va servir comme entrée pour l'outil *a2cpp* permettant de générer le code *C++* correspondant (modélisant le fonctionnement du pipeline).

Dans un premier temps, nous allons expliquer la génération des classes d'instructions. Ensuite, nous allons nous intéresser à la génération des classes d'instructions de dépendance de données. De plus, le fonctionnement du contrôleur de dépendance de données sera explicité. Ensuite, nous allons présenter le principe sur lequel est basée l'exécution des instructions à travers différents étages du pipeline dans les deux cas : un seul pipeline ou plusieurs pipelines. Enfin, quelques résultats sur le processus de génération du simulateur à partir de plusieurs descriptions de

1. Un exemple de ce fichier est donné dans l'annexe C.

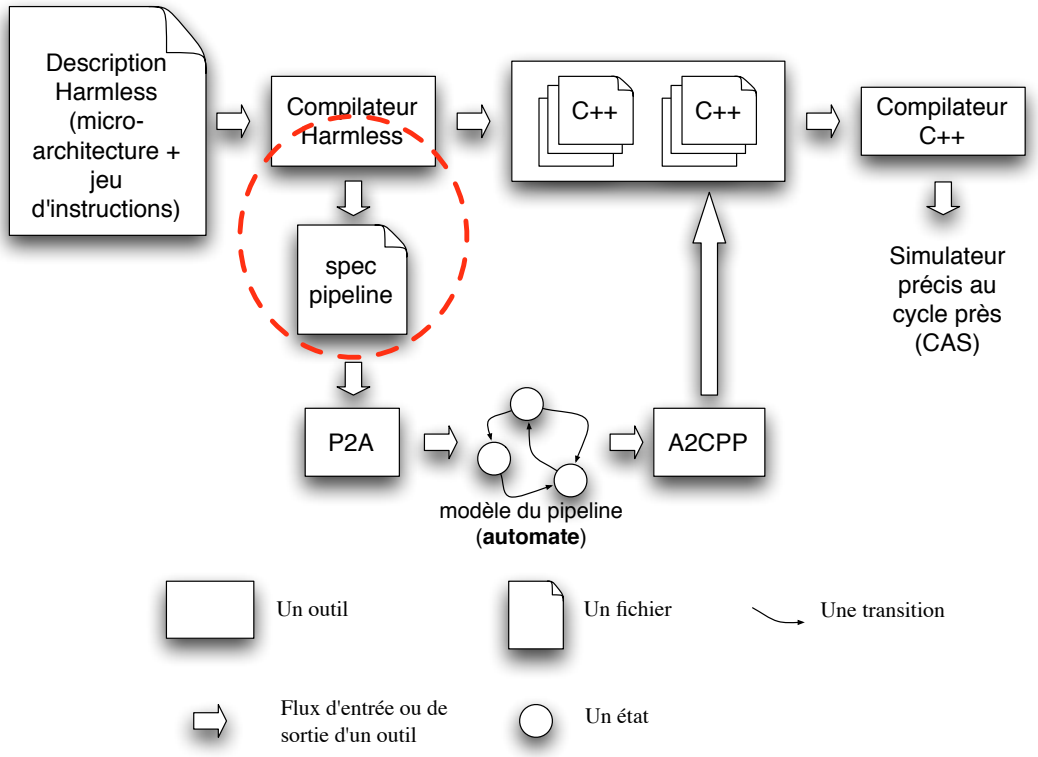



FIGURE 7.1 – Chaîne de développement du simulateur *CAS*. Ici, nous nous intéressons à la partie entourée par un cercle pointillé (c'est-à-dire la partie du compilateur s'occupant de la génération du fichier spécifiant le pipeline décrit dans HARMLESS).

processeurs seront interprétés et analysés.

## 7.2 Génération des classes d'instructions

À partir de la vue `micro-architecture` et de la vue `behavior`, les instructions sont regroupées en classes pour réduire l'espace d'état d'automate (modèle interne du pipeline). D'après la définition dans la section 5.3.2, une classe d'instructions rassemble les instructions qui utilisent les mêmes ressources (internes et externes) le même nombre de fois. Dans notre approche, les ressources externes correspondent aux ports partagés et les autres ports forment les ressources internes.

```
Pour Chaque section architecture faire
    Pour Chaque instruction et chaque behavior faire
        - Trouver toutes les méthodes des composants utilisées par cette instruction ;
        - Trouver les ports correspondant à ces méthodes ;
    Fin Pour
    Pour Chaque pipeline mappé sur l'architecture courante faire
        - Dans chaque étage du pipeline, trouver les ports correspondant aux méthodes utilisées des composants ;
        - Vérifier le mappage de la vue sémantique du jeu d'instructions sur le pipeline à travers les ports ;
        Si (Le mappage est réussi) Alors
            - Créer une table vide qui va contenir les classes d'instructions pour le pipeline courant ;
            - Créer la première classe d'instructions (elle va contenir les instructions n'utilisant aucune ressource comme l'instruction NOP) et l'ajouter à la table ;
            Pour Chaque instruction faire
                - Créer et initialiser une variable booléenne : trouvé = faux ;

                Pour Chaque classe d'instructions faire
                    Si (Cette instruction utilise les mêmes ports le même nombre de fois que la classe d'instructions courante) Alors
                        - Ajouter l'instruction à cette classe d'instructions ;
                        - trouvé = vrai ;
                    Fin Si
                Fin Pour
                Si (trouvé == faux) Alors
                    - Créer une nouvelle classe d'instructions et l'ajouter à la table ;
                    - Ajouter l'instruction à cette nouvelle classe d'instructions ;
                Fin Si
            Fin Pour
        Sinon
            - Envoyer un message d'erreur ;
        Fin Si
    Fin Pour
Fin Pour
```

Algorithme 3: L'algorithme simplifié permettant de construire les classes d'instructions.

Le principe de génération des classes d'instructions est présenté dans l'algorithme simplifié 3. L'idée de base de cet algorithme est de parcourir la vue `behavior` et de retrouver, pour chaque instruction, les différentes méthodes qu'elle utilise pour accéder aux composants, pour ensuite les faire correspondre aux ports définis dans une sous-vue `architecture` donnée. De même, pour chaque sous-vue `pipeline` associée à cette architecture, il faut trouver les ports correspondant aux méthodes des composants utilisées dans chacun de ses étages. Et après vérification que toutes les instructions se mappent correctement sur les différents étages du pipeline, les instructions utilisant les mêmes ports le même nombre de fois seront regroupées dans une classe d'instructions pour un pipeline donnée. Les mêmes étapes sont réitérées pour chaque sous-vue `architecture` définie dans la description.

Reprenons l'exemple de description de la micro-architecture dans HARMLESS présenté dans la section 6.3. Cet exemple décrit un pipeline `sample_pipe` de 6 étages mappé sur une architecture `Generic`, également définie. En utilisant un modèle basé sur le jeu d'instructions du co-processeur *RISC XGate* du *HCS12X* de *FreeScale* [19], avec ce pipeline de 6 étages, 15 classes d'instructions sont générées en appliquant l'algorithme 3.

**Réduction des classes d'instructions** Comme nous l'avons déjà montré, la taille de l'automate (le modèle d'exécution du pipeline) augmente polynomialement avec le nombre de classes d'instructions, ce qui peut être la cause d'une explosion combinatoire (voir section 5.3.4.4). Il est donc intéressant de réduire au maximum leur nombre.

La réduction du nombre de classes d'instructions est possible en diminuant le nombre de ressources internes (le nombre de ressources externes ne peut pas être réduit parce qu'elles peuvent être prises par d'autres composants matériels qui évoluent concurremment, comme un *contrôleur mémoire* par exemple, ces ressources décrivent des contraintes dynamiques qui seront résolues pendant la simulation), et ceci en supprimant les ports privés (non partagés) qui ne représentent pas des contraintes sur l'utilisation des méthodes de composants, comme le montre l'algorithme 4. Par exemple, si dans un processeur donné, les registres sont en mesure d'autoriser, en parallèle, 3 accès en lecture et qu'avec n'importe quelle combinaison d'instructions, en état d'exécution dans le pipeline, l'accès aux registres ne dépasse pas le nombre autorisé (ici 3), ce port ne forme pas vraiment une ressource interne, et nous pouvons le supprimer. Dans cet algorithme, nous distinguons deux cas de figure pour une raison de performance :

– cas où le nombre d'apparition d'une ressource interne, dans les étages du pipeline, est égal à 1 (ressource prise dans un seul étage du pipeline), cette ressource peut être supprimée si son utilisation, par toutes les instructions

la nécessitant, ne dépasse pas le nombre d'utilisations autorisées dans la sous-vue `architecture` ;
- cas général où il sera nécessaire de tester toutes les combinaisons possibles d'instructions (leur nombre est donné par l'arrangement mathématique $A_c^n$ où $n$ est le nombre d'étages du pipeline et $c$ le nombre de classes d'instructions), en état d'exécution dans le pipeline, pour voir si, à un instant donné, le nombre d'utilisation spécifié dans la sous-vue `architecture` sera dépassé.

Reprenons le même exemple de la section 6.3 décrivant la micro-architecture composée du pipeline `sample_pipe` de 6 étages mappé sur l'architecture `Generic`. En utilisant le modèle basé sur le jeu d'instructions du co-processeur *RISC XGate* du *HCS12X* de *FreeScale* [19], les 15 classes d'instructions, construites en appliquant l'algorithme 3, sont réduites en 6 classes d'instructions en exécutant l'algorithme 4. En effet, avant la réduction, 4 ports (non partagés) sont nécessaires :

- 1 port pour la gestion de l'Unité Arithmétique et Logique : `alu` ;
- 2 ports pour contrôler l'accès aux registres `rs` (autorisant au maximum 3 lectures en parallèle) et `rd` (autorisant au maximum 3 écritures en parallèle) ;
- 1 port pour la gestion du branchement : `branch` ;

Après l'exécution de l'algorithme 4, seulement 1 port s'avère nécessaire : le port `alu`, réduisant ainsi le nombre de classes d'instructions de 15 classes à 6 classes (ces classes sont présentées dans l'annexe C).

## 7.3 Génération des classes d'instructions de dépendance de données

De la même manière, la génération des classes d'instructions de dépendance de données (définies dans la section 5.3.3) se fait, pour une description du processeur donnée, à partir de la vue `micro-architecture` et la vue `behavior`. En effet, pour chaque sous-vue `architecture` définie dans la description de la micro-architecture et pour chaque `pipeline` mappé sur cette `architecture`, les instructions, qui utilisent les même ports permettant de contrôler l'accès en lecture et/ou en écriture aux registres dans le même étage du pipeline, sont regroupées dans une classe d'instructions de dépendance de données.

En utilisant le même exemple que dans la section 7.2, 5 classes d'instructions de dépendance de données sont obtenues :

- une classe qui regroupe les instructions qui lisent un registre dans l'étage `Decode` du pipeline ;
- une classe qui regroupe les instructions qui lisent un registre dans l'étage `Decode` et écrivent dans un registre au cours de l'étage `Register` ;

```
Pour Chaque section architecture faire
    Pour Chaque pipeline associé à cette architecture faire
        - Créer une table vide ;
        Pour Chaque port non partagé utilisé dans ce pipeline faire
            - Compter le nombre d'apparitions de ce port dans les étages du
            pipeline ;
            - Mémoriser le port et son nombre d'apparitions dans la table ;
        Fin Pour
        Pour Chaque port stocké dans la table faire
            - Récupérer son nombre d'apparitions dans les étages du pipeline ;

            Si (Ce nombre == 1) Alors
                Pour Chaque classe d'instructions faire
                    - Créer et initialiser une variable booléenne :
                    contrainte = faux ;
                    - Compter le nombre d'utilisations de ce port ;
                    Si (Le nombre d'utilisations dans cette classe > Le
                    nombre d'utilisations autorisé dans cette architecture)
                    Alors
                        - contrainte = vrai ;
                    Fin Si
                Fin Pour
                Si (contrainte == faux) Alors
                    - Supprimer ce port de toutes les classes d'instruc-
                    tions qui l'utilisent parce qu'il ne représente pas une
                    contrainte d'accès aux composants ;
                Fin Si
            Sinon
                - Tester toutes les combinaisons possibles de classes d'ins-
                tructions, en état d'exécution dans le pipeline ;
                Si (Pour toutes les combinaisons, le nombre d'utilisations
                de ce port dans ce pipeline ⩽ nombre d'utilisations autorisé
                dans cette architecture) Alors
                    - Supprimer ce port de toutes les classes d'instruc-
                    tions qui l'utilisent parce qu'il ne représente pas une
                    contrainte d'accès aux composants ;
                Fin Si
            Fin Si
        Fin Pour
        - Reconstruire les classes d'instructions ;
    Fin Pour
Fin Pour
```

Algorithme 4: L'algorithme simplifié permettant de réduire les classes d'instructions.

- une classe qui regroupe les instructions qui écrivent dans un registre au cours de l'étage `Register` ;
- une classe qui regroupe les instructions qui lisent un registre dans l'étage `Decode` ou l'étage `Memory` ;
- une classe qui regroupe les instructions qui lisent un registre dans l'étage `Decode` ou l'étage `Memory` et écrivent dans un registre au cours de l'étage `Register`.

## 7.4 Contrôleur de dépendance de données

Une dépendance de données se produit quand une instruction a besoin de lire un registre qui sera mis à jour, plus tard dans le pipeline par une instruction précédent celle-ci.

Pendant la simulation, pour contrôler l'accès aux registres (en écriture ou en lecture), un *contrôleur de dépendance de données* est utilisé comme le montre la figure 7.2. Comme nous l'avons déjà dit, une des propriétés des ressources externes est qu'elle permet de résoudre les aléas de données. Une telle ressource externe est générée pour chaque étage de pipeline où un accès en lecture aux registres est nécessaire et est associée au contrôleur de dépendance de données.

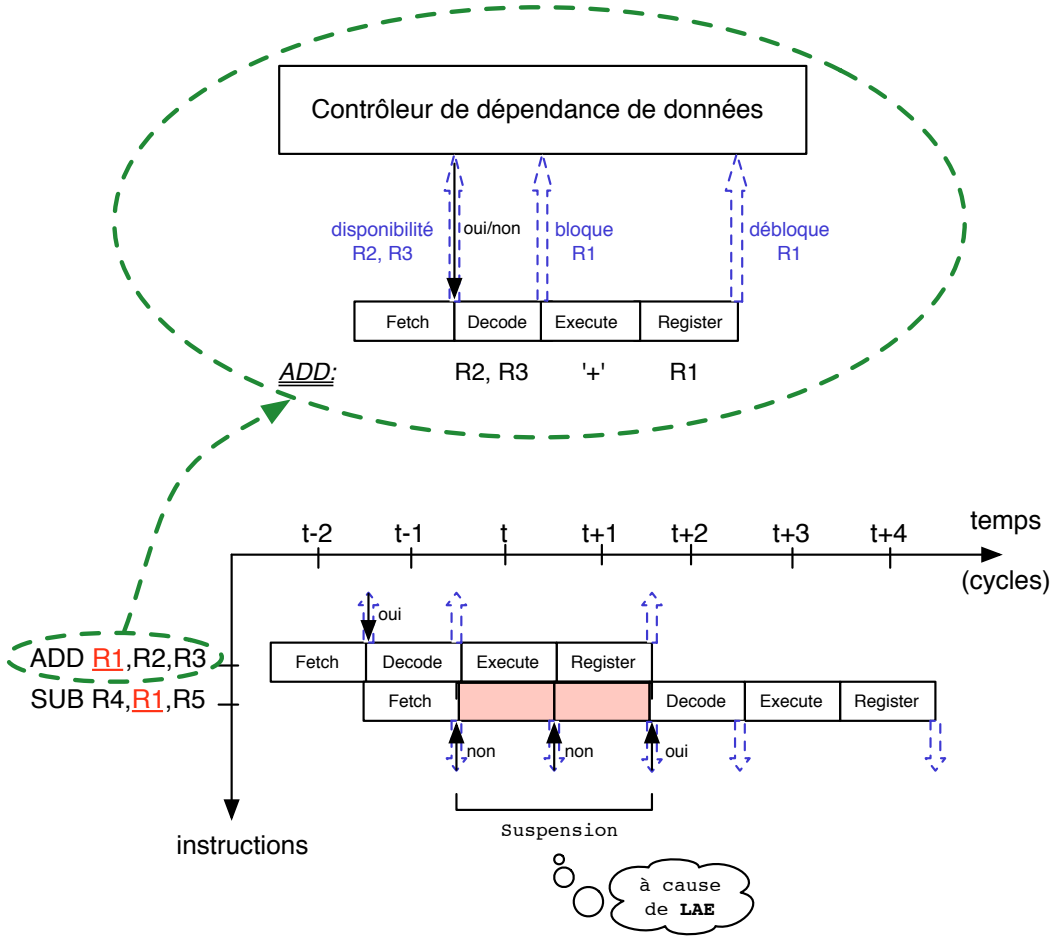



FIGURE 7.2 – Interaction entre le contrôleur de dépendance de données et les deux instructions : *ADD R1, R2, R3* et *SUB R4, R1, R5*, durant la simulation. L'instruction *SUB* a besoin de *R1* et *R5* (registres sources) pour entrer dans l'étage `Decode`. Or, le registre *R1* sera mis à jour par l'instruction précédente *ADD* dans l'étage `Register`, donc la requête au Contrôleur de dépendance de données échoue. L'instruction *SUB* sera bloquée dans l'étage `fetch` jusqu'à ce que l'instruction *ADD*, écrive le résultat dans l'étage `Register`.

Comme modèle du pipeline nous supposons que les deux premiers étages sont l'étage *Fetch*, suivi de l'étage *Decode* qui lit les opérandes, et que dans le dernier étage *Register* fait l'accès en écriture aux registres. Le contrôleur de dépendance de données fonctionne comme présenté dans l'algorithme simplifié 5. En effet, lorsque les instructions sont exécutées dans le pipeline, l'instruction qui est dans

l'étage *Fetch* envoie une requête au contrôleur pour vérifier si tous ses opérandes, nécessaires dans le prochain étage (*Decode*), sont disponibles. Si au moins un des registres est occupé (car il est utilisé par une autre instruction dans le pipeline), la demande échoue et la ressource externe associée est définie comme *non disponible*. De cette manière, quand la condition de la transition est évaluée pour obtenir l'état suivant de l'automate modélisant le fonctionnement du pipeline, la transition associée à cette nouvelle condition (la condition de la transition dépend de l'état des ressources externes) conduira à un état qui insère une bulle dans le pipeline permettant ainsi à l'instruction qui est dans l'étage *Fetch* d'attendre ses opérandes, et résoudre la dépendance de données.

**Si** (Il y a une instruction dans l'étage `Fetch` **et** Cette instruction aura besoin d'opérandes dans l'étage `Decode`) **Alors**
- L'instruction envoie une requête au contrôleur ;
- **Si** (Au moins un opérande n'est pas disponible) **Alors**
  - La requête échoue : la ressource externe associée passe à l'état `non disponible`) ;
- **Sinon**
  - La requête est validée : la ressource externe associée passe à l'état `disponible`) ;
- **Fin Si**

**Sinon**
- La requête est validée : la ressource externe associée passe à l'état `disponible`) ;

**Fin Si**
**Si** (Il y a une instruction dans l'étage `Decode` **et** Cette instruction va écrire un résultat dans l'étage `Register`) **Alors**
- L'instruction envoie une requête au contrôleur ;
- Le contrôleur bloque le registre qui sera mis à jour ;

**Fin Si**
**Si** (Il y a une instruction dans l'étage `Register` **et** Cette instruction écrit un résultat) **Alors**
- L'instruction envoie une requête au contrôleur ;
- Le contrôleur débloque le registre mis à jour ;

**Fin Si**

Algorithme 5: Interaction entre le contrôleur de dépendance de données et une instruction durant la simulation.

## 7.5 Exécution des instructions

Comme nous l'avons déjà dit, la description fonctionnelle est indépendante de la description de la micro-architecture. D'où, pour chaque instruction et à partir de la vue sémantique, une seule fonction d'exécution (comme expliqué dans la section 4.2), au lieu d'une fonction par étage du pipeline comparé aux autres *ADL*s, est générée dans le simulateur *CAS*.

En effet, à partir de l'état courant de l'automate à états finis (modélisant le fonctionnement interne du pipeline), une fonction permet, tout d'abord, de vérifier les aléas de données en interrogeant le contrôleur de dépendance de données. Après, le code de l'instruction à exécuter va être lu en mémoire, et son code binaire décodé. Ensuite, à partir de l'état courant de l'automate, de la classe d'instructions de cette instruction et l'état des ressources internes et externes, l'état suivant de l'automate sera obtenu. À son entrée dans le pipeline, l'instruction sera exécutée totalement. Le terme « totalement » signifie ici que les instructions s'exécutent entièrement (sans notion de temps), et ensuite l'automate à états finis, simule leur passage à travers les différents étages du pipeline. Ceci permet d'augmenter la vitesse de la simulation.

Cependant, comme le montre la figure 7.3, nous n'utilisons pas des tampons, les registres sont mis à jour directement au moment de l'exécution de l'instruction (c'est-à-dire, au début du pipeline), donc nous ne pouvons pas avoir l'état exact des registres à un instant donné. Mais ça n'est pas gênant vis à vis de l'exécution car elle conserve la sémantique du programme. En outre, l'utilisation des tampons permet l'exécution de l'accès mémoire (ou autre) à la date précise (sans tampon, les accès sont effectués au moment du fetch). Avec notre approche, nous pouvons basculer rapidement et dynamiquement entre le simulateur *ISS* et le simulateur *CAS*. Toutefois, l'ajout des tampons va être envisagé dans les travaux futurs pour que les accès mémoire soient faits à la date exacte (important dans le cas d'un système multiprocesseur, par exemple).

**Cas d'un pipeline découpé en plusieurs pipelines** Dans le cas d'un pipeline découpé en plusieurs pipelines de plus petite profondeur, l'instruction sera exécutée totalement avant de passer à travers les étages des différents pipelines. Le passage entre les différents pipelines se passe comme le montre la figure 7.4.

Sur cette figure, les 3 pipelines sont ceux décrits dans la section 6.5. Cette figure montre l'état des 3 pipelines à l'instant *t* et à l'instant *t+1*. Les flèches pointillées indique le sens et la possibilité de déplacement des instructions d'un étage à un autre. La forme carrée avec double cadre est un tampon à une seule entrée. À chaque fois qu'une instruction arrive au dernier étage d'un pipeline (sauf dans le cas du dernier pipeline), elle sera mémorisée dans ce tampon. Si le premier étage du pipeline suivant est capable de recevoir une instruction, il va chercher l'instruction

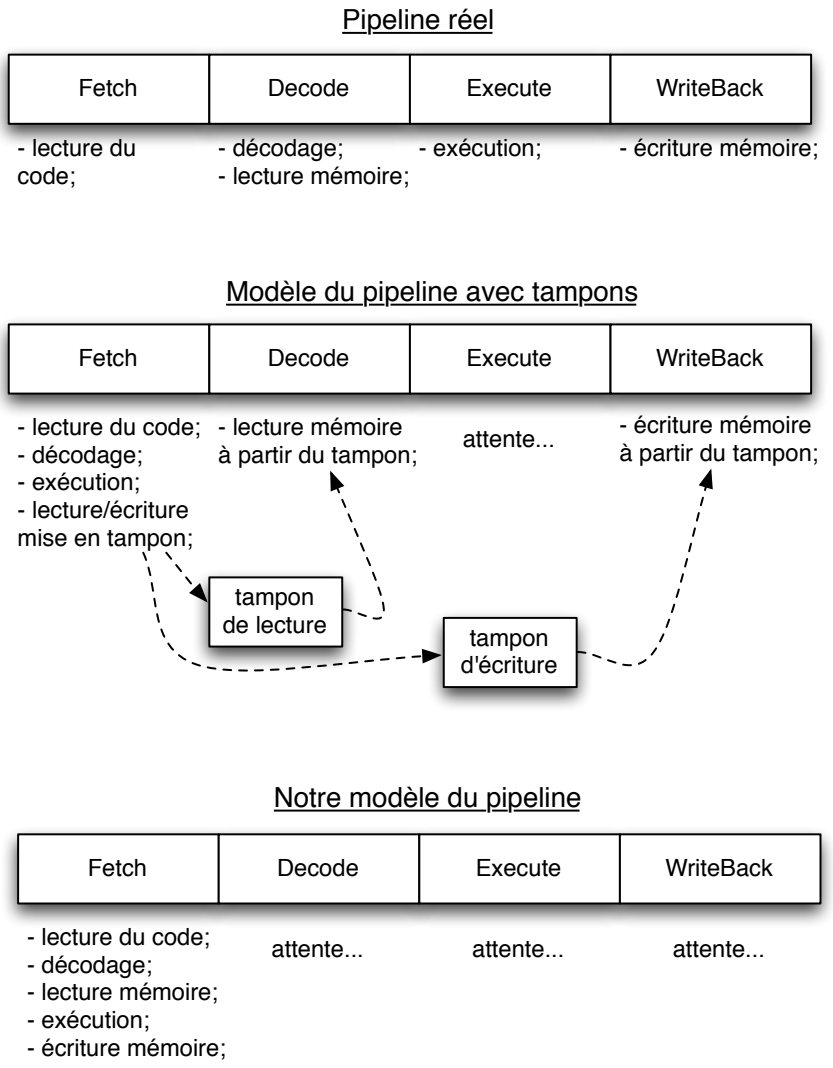



FIGURE 7.3 – Notre approche d'exécution dans un pipeline à 4 étages. Dans cet exemple, 3 cas sont présentés : cas d'un pipeline réel, modèle d'un pipeline utilisant des tampons afin de conserver l'état exact des registres à chaque instant et exécuter l'accès mémoire à la date précise et notre modèle permettant de basculer dynamiquement entre les deux simulateurs : le *ISS* et le *CAS*.

directement dans ce tampon.

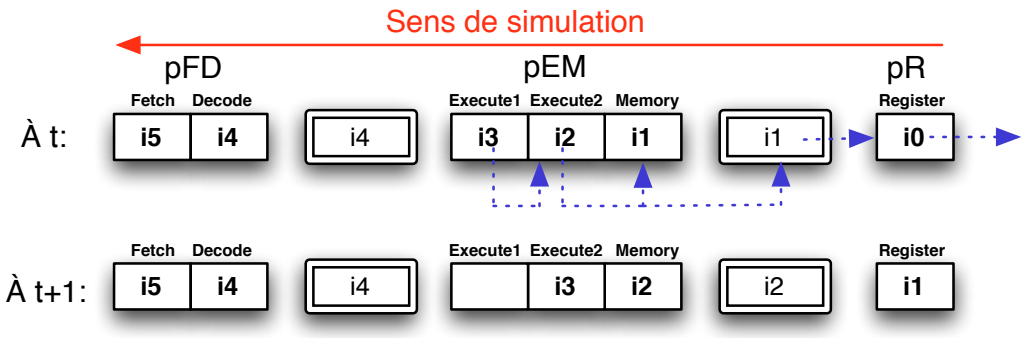



FIGURE 7.4 – Passage des instructions d'un étage à un autre à travers les différents pipelines pendant la simulation. Supposons que le calcul dans l'Unité Arithmétique et Logique (*UAL*) nécessite 2 étages (*Execute1* et *Execute2*) et n'est pas pipeliné. Supposons aussi que l'instruction *i3* utilise l'*UAL* et l'instruction suivante *i4* en a besoin également. Pour cette raison, à l'instant *t+1*, l'instruction *i4* n'a pas pu entrer dans l'étage *Execute1* du pipeline *pEM* et le pipeline *pFD* est resté dans le même état.

## 7.6 Expérimentation et Analyse des résultats

Cette section présente quelques résultats sur le processus de génération et d'exécution du simulateur *CAS* ainsi qu'une comparaison avec une cible réelle. Elle donne le temps de calcul nécessaire pour simuler un exemple simple basé sur un exécutif temps réel (*Trampoline*[2]) sur le simulateur d'une part et sur la cible réelle d'autre part. L'exemple utilisé se compose de 3 tâches et engendre de nombreuses préemptions. L'essentiel du code est composé des appels système du *RTOS*. Le processeur utilisé est le *PowerPC 5516* de chez *Freescale* [21]. Le jeu d'instructions et le pipeline de base des modèles sont donc ceux du processeur *PowerPC 5516*. Pour ce processeur, le pipeline est composé de 4 étages :

- *Fetch* : permet de récupérer le code de l'instruction dans la mémoire ;
- *Decode* : s'occupe du décodage de l'instruction, de la lecture des opérandes, d'exécution des instructions de branchement et du calcul des adresses effectives pour les opérations de la mémoire ;
- *Execute* : s'occupe de l'exécution de l'instruction et permet d'accéder à la mémoire ;
- *Write Back* : permet d'écrire le résultat dans le banc des registres. Un court-

2. *Trampoline* est un *RTOS* distribué sous forme de logiciel libre, conforme au standard *ISO OSEK/VDX* (`http://trampoline.rts-software.org/`).

circuit permet d'envoyer le résultat à l'étage *Execute* du pipeline.

### 7.6.1 ISS *versus* CAS

Cette section présente un certain nombre de résultats dans le tableau 7.1 :

**ISS** : cette colonne donne les performances du simulateur de type *ISS* pour le *PowerPC 5516*;

**CAS4** : pour ce modèle, la micro-architecture (le pipeline et les concurrences d'accès aux composants matériels) du processeur *PowerPC 5516* est également modélisée. Cette colonne présente les performances du simulateur de type *CAS* pour ce processeur ;

**CAS5 et CAS6** : ces deux colonnes présentent les performances de deux variantes fictives de pipeline mettant ainsi en avant l'intérêt de la séparation des vues micro-architecture et jeu d'instructions. Elles modélisent respectivement des pipelines de 5 et 6 étages. Dans ces deux cas, l'étage *Execute* a été respectivement divisé en 2 et 3 étages. Ceci se fait en modifiant quelques lignes dans la sous-vue `pipeline` de la description (6 lignes sont ajoutées pour pouvoir passer du modèle *CAS4* au modèle *CAS6*). Cependant, nous pouvons remarquer que ceci impacte la taille de l'automate généré (explosion combinatoire) ;

**CAS6 avec découpage** : pour pallier cette explosion combinatoire, fonction notamment de la taille du pipeline, cette dernière colonne présente le modèle avec le pipeline à 6 étages, mais divisé en trois pipelines de plus faible profondeur (chacun formé de deux étages).

Nous pouvons remarquer que le processus de génération du simulateur est très rapide : le simulateur est construit à partir du modèle, sur un processeur *Intel Core 2 Duo* à 2 GHz avec 2 Go de RAM, en moins de 60 s dans tous les cas (hors cas du modèle *CAS6 avec découpage* qui sera analysé un peu plus loin), même pour le pipeline à 6 étages qui nécessite un plus grand automate.

Mais comme ce processus de génération est exécuté une seule fois pour chaque modèle, le point le plus important à comparer concerne la vitesse de simulation. Elle se réfère aux quatre dernières lignes du tableau. Nous pouvons voir sur le tableau 7.1 que les temps de calcul nécessaires pour les trois modèles (*CAS4*, *CAS5* et *CAS6*) sont dans le même ordre de grandeur. Pour un nombre d'instructions donné, le simulateur avec le plus court pipeline (*CAS4*) est le plus rapide, il exige également moins de cycles pour le même travail. Par exemple, 25.25 s sont nécessaires pour simuler 100 millions d'instructions (157 millions de cycles). A titre de comparaison, l'*ISS* nécessite 3.58 s pour le même scénario. Le rapport de surcoût de calcul des propriétés temporelles par rapport à l'exécution fonctionnelle est compris entre 7.05 et 8.94, en fonction de la profondeur des pipelines.

| | **ISS** | **CAS4** | **CAS5** | **CAS6** | **CAS6 avec découpage** | | |
|---|---|---|---|---|---|---|---|
| Longueur de description (lignes) | 3208 | 3282 | 3285 | 3288 | 3292 | | |
| Temps pour générer le code source de l'ISS | 4,3s | | | | | | |
| Temps pour générer le code source du modèle de pipeline | - | 1.83s | 1.82s | 1.83s | 2.87s | | |
| Temps pour générer le code source du CAS | - | 6.13s | 6.12s | 6.13s | 7.17s | | |
| Nombre d'états de l'automate | - | 43 | 111 | 289 | 7 | 3 | 3 |
| Nombre de transitions dans l'automate | - | 111 | 289 | 755 | 17 | 3 | 3 |
| Taille du code source du simulateur (lignes C++) | 40970 | 51055 | 51160 | 51368 | 59644 | | |
| Temps de compilation | 32.38s | 53.35s | 53.39s | 53.42s | 86.69s | | |
| Execution de 100 millions d'instructions | *3.58s* | *25.25s* | *29.29s* | *32.04s* | *114.95s* | | |
| Surcoût CAS/ISS | - | *7.05* | *8.18* | *8.94* | *32.1* | | |
| CPI | - | *1.57* | *2.04* | *2.52* | | | |
| Execution de 100 millions de cycles | - | *16.13s* | *14.55s* | *12.66s* | *43.21s* | | |

TABLE 7.1 – Ce tableau présente quelques résultats sur le processus de génération des simulateurs *ISS* et *CAS* et leurs temps d'exécution. Les mesures de temps ont été effectuées sur un processeur *Intel Core 2 Duo* à 2 GHz avec 2 Go de RAM.

Un autre point intéressant est que l'exécution de 100 millions de cycles d'horloge est un peu plus rapide sur le pipeline le plus profond (*CAS6*). Ce résultat peut s'expliquer par le fait qu'il y a moins d'instructions qui sortent du pipeline (débit d'instructions) dans ce modèle (la pénalité des aléas de données augmente avec la profondeur du pipeline). Enfin, nous pouvons remarquer que la surcharge de calcul induite par l'augmentation du nombre d'étages de pipeline est extrêmement faible,

en raison de l'approche interne basée sur un automate à états finis.

**Impact du découpage de pipeline du modèle *CAS6*** D'après les résultats des deux dernières colonnes (*CAS6* et *CAS6 avec découpage*), nous pouvons constater que le découpage du pipeline a comme effet d'une part de réduire l'espace d'état des automates en générant 3 automates à états finis plus petits permettant ainsi de résoudre le problème de l'explosion combinatoire (bien qu'ici il n'y en ait pas). D'autre part, il induit une perte de performance en simulation (un surcoût d'un rapport de 3.58 : 114.95 s pour le modèle *CAS6* avec un pipeline découpé en 3 pipelines de deux étages chacun contre 32.04 s pour le modèle *CAS6*).

**Impact du contrôleur de dépendance de données sur les performances en simulation** Si nous supprimons le rôle du contrôleur de dépendance de données dans notre simulateur, nous remarquons un gain de performance de 48.5% (le temps d'exécution de 100 millions d'instructions passe de 25.25 s à 13 s pour le modèle *CAS4*). Par contre, les résultats temporels obtenus n'ont aucune signification car ils correspondent à un fonctionnement sans prendre en compte les aléas de données. Toutefois, ces résultats permettent de mettre en évidence la contribution du contrôleur de dépendance de données sur le temps total nécessaire pour la simulation. Ceci nous encourage, dans les travaux futurs, à optimiser ce contrôleur.

### 7.6.2 Comparaison avec la cible réelle

Cette section compare les performances du modèle *CAS4* (en nombre de cycles d'horloge) avec la cible réelle *PowerPC 5516*. La fréquence d'horloge du processeur utilisé pour le test est de 16MHz (cycle d'horloge de $6.25 \times 10^{-8}$ $s$).

Deux mesures ont été réalisées sur le processeur réel (2.67 $ms$ pour la première et 85 $\mu s$ pour la deuxième). Le tableau 7.2 donne un aperçu des performances du simulateur par rapport à la cible réelle. Nous pouvons remarquer que les performances du simulateur *CAS*, généré à partir de la description du processeur *PowerPC 5516* en HARMLESS, sont bonnes et très proches de la réalité : l'erreur obtenue est de moins de 5% pour les deux mesures, ce qui est largement acceptable.

## 7.7 Conclusion

Dans ce chapitre, nous avons présenté les différentes étapes de la génération du simulateur précis au cycle près (*CAS*). Pour obtenir un tel simulateur, la description fonctionnelle du jeu d'instructions et la description de la micro-architecture du processeur sont nécessaires. Tout au long de ce chapitre, nous avons expliqué la construction des classes d'instructions et la méthode permettant leur réduction

| | **Nombre de cycles requis pour la 1re mesure (25919 instructions)** | **CPI** | **Nombre de cycles requis pour la 2e mesure (837 d'instructions)** | **CPI** |
|---|---|---|---|---|
| Modèle | *40706* | *1.57* | *1308* | *1.56* |
| Cible réelle | *42750* | *1.65* | *1360* | *1.62* |
| Pourcentage d'erreur | 4.8% | | 3.8% | |

TABLE 7.2 – Ce tableau compare le CPI du modèle *CAS4* avec celui de la cible réelle, le *PowerPC 5516*. Pour le modèle, les mesures de temps ont été effectuées sur un processeur *Intel Core 2 Duo* à 2 GHz avec 2 Go de RAM.

afin de diminuer la taille de l'automate obtenu et éviter l'explosion combinatoire. De plus, nous avons présenté la génération du contrôleur de dépendance de données et des classes d'instructions de dépendance de données. Nous avons aussi signalé que l'exécution des instructions se fait totalement avant de simuler leur passage à travers différents étages du pipeline. Enfin, les résultats obtenus montre que la performance du simulateur *CAS* est bonne.

# Chapitre 8

# Conclusion générale et perspectives

L'architecture matérielle des systèmes embarqués temps réel devient de plus en plus sophistiquée et complexe, ce qui entraîne l'augmentation des besoins des outils de validation de plus en plus fine (*i.e.* proche du système réel). En effet, de tels systèmes, en raison de leur criticité élevée, peuvent causer, en cas de défaillance, des dommages plus ou moins graves tant au niveau économique qu'humain. Plusieurs approches de validation, souvent complémentaires, existent telles que l'analyse formelle (réseaux de Petri T-temporisé, l'analyse d'ordonnançabilité) et la simulation (au niveau système d'exploitation ou matériel).

Les travaux que nous avons présentés dans ce mémoire portent sur l'analyse des systèmes embarqués temps réel via la simulation de leur architecture matérielle, et plus particulièrement l'élément constituant le cœur de cette architecture : le processeur. En effet, il joue un rôle important sur les caractéristiques temporelles de tels systèmes.

Le développement « manuel » d'un simulateur du support matériel s'avère lent, difficile et sujet aux erreurs, surtout pour les processeurs à architecture complexe. Un premier objectif visé dans le cadre de cette thèse était de pouvoir générer un tel simulateur automatiquement en utilisant un langage de description d'architecture matérielle (*ADL*). Cet ADL doit permettre d'une part la description du jeu d'instructions pour engendrer un simulateur fonctionnel (*ISS*), et d'autres part la description de la micro-architecture (*i.e.* le pipeline et les concurrences d'accès aux différents composants matériels) pour engendrer un simulateur précis au cycle près (*CAS*). Nous proposons ici un nouvel *ADL* « HARMLESS » qui a été conçu avec les objectifs suivants :

– Une description *permettant la vérification*. En effet, HARMLESS offre les avantages suivants : un typage fort des variables, pas de possibilité d'inclure des parties décrites directement en une langage de programmation comme le

*C*) ;
- Une description *concise* et *incrémentale* autorisant une description incomplète du jeu d'instructions ;
- Une description fonctionnelle du jeu d'instructions *indépendante de la micro-architecture* du processeur permettant ainsi d'engendrer séparément ou conjointement les deux types de simulateurs : *ISS* et *CAS*.

Une première étape a été donc la génération d'un simulateur fonctionnel à partir de la description du jeu d'instructions et la description fonctionnelle des composants matériels en utilisant HARMLESS. La description du jeu d'instructions se fait à travers trois vues séparées : la vue binaire (`format`) utilisée pour décrire le format binaire des instructions (au niveau du simulateur, elle permet le décodage des instructions), la vue comportementale (`behavior`) utilisée pour décrire le comportement des instructions (au niveau du simulateur, elle permet l'exécution des instructions) et la vue syntaxique (`syntax`) utilisée pour décrire la syntaxe textuelle de l'instruction (au niveau du simulateur, elle permet le désassemblage des instructions). La description fonctionnelle des composants matériels se base sur le principe d'encapsulation, et est séparée de la description du jeu d'instructions pour pouvoir générer en plus un simulateur précis au cycle près.

Une seconde étape a consisté en la génération d'un simulateur précis au cycle près nécessitant l'ajout d'une quatrième vue : la vue micro-architecture. La description de la vue micro-architecture se fait à travers 3 sous-vues : la sous-vue `architecture` permettant la description des concurrences d'accès aux différents composants matériels (les registres, les unités fonctionnelles, la mémoire, ...), la sous-vue `pipeline` permettant de décrire un pipeline et la sous-vue `machine` permettant de décrire l'architecture pipelinée qui peut contenir un ou plusieurs pipelines (cas d'un pipeline découpé). La modélisation interne du pipeline est basée sur un automate à états finis. Or, pour une architecture complexe, le problème de l'explosion combinatoire peut survenir. Pour résoudre ce problème, un découpage du pipeline en plusieurs pipelines de plus faibles profondeurs permet de générer des automates plus petits. Enfin, HARMLESS offre la possibilité de décrire la prédiction de branchement dynamique, utilisant un compteur à 1 ou 2 bits.

Ces deux étapes ont été implémentées dans un outil HARMLESS permettant de générer simultanément les deux simulateurs *ISS* et *CAS*. Les performances de cet outil ont été testées vis à vis d'une cible réelle : le processeur PowerPC 5516. Un pourcentage d'erreur de moins de 5% a été constaté.

# Perspectives

Les différents résultats obtenus sur le processus de génération et d'exécution des simulateurs fonctionnels et/ou précis au cycle près de plusieurs processeurs, tels

que le *PowerPC 5516*, le *HCS12* et son co-processeurs le *XGate*, l'*AVR*, sont bons. Ces tests montrent que HARMLESS permet d'obtenir un simulateur performant du support matériel dans le but de valider une architecture matérielle pour une application temps réel. Plusieurs extensions, à plus ou moins long terme, permettent d'améliorer les performances du simulateur (en temps de simulation) d'une part et d'autre part offre la possibilité de décrire un plus large spectre d'architectures des processeurs existants.

Pour le moment, la description du *BTB* et de la politique de prédiction de branchements sont faites directement dans le composant chargé de la gestion du compteur programme.

À court terme, il faudrait les déplacer et les mettre dans la vue micro-architecture afin de bien respecter l'objectif de HARMLESS : *une description fonctionnelle du jeu d'instructions indépendante de la micro-architecture*. D'autre part, il serait intéressant d'étendre l'expressivité du langage à l'ensemble des éléments architecturaux de branchements. De plus, il serait indispensable de pouvoir décrire les différents niveaux de mémoire constituant la hiérarchie mémoire. En outre, il faudrait ajouter le support des périphériques (comme les timers) pour permettre la simulation des événements périodiques dans les systèmes d'exploitation temps réel.

À plus long terme, il serait nécessaire d'étendre la description de la micro-architecture pour pouvoir prendre en compte des architectures encore plus complexes, tel que l'architecture superscalaire [1] ; ceci a été sommairement abordé dans la section 6.5.

1. L'ensemble des travaux détaillés ici devrait être compatible avec cette extension.

# Annexe A

# Simulateur de jeu d'instructions

## A.1 Exemple sur la description de la vue binaire

Cet exemple présente comment décrire une partie du jeu d'instructions du processeur *XGate*, ainsi que les fichiers générés. Le *XGate* est un co-processeur intégré dans le microcontroleur *HCS12X*. Il est basé sur une architecture *RISC*, avec une taille d'instruction fixe de 16 bits. Il dispose de 16 registres de 8 bits.

### A.1.1 Description d'un sous ensemble d'instructions du XGate

Le tableau A.1 donne le format binaire des instructions de décalage (*shift*) et les instructions triadiques logiques (3 registres).

Nous pouvons directement remarquer sur cet exemple que le bit 12 sert à différencier les 2 types d'instructions. D'une manière plus générale, sur tout le jeu d'instructions, les 5 bits de poids forts permettent d'identifier les familles d'instructions (*opcode*) :

```
1 format inst
2   select slice {15..11}
3     case \b00001 is shiftInstructionImm
4     case \b00010 is logicalTriadic
5   end select
6 end format
```

| **Fonctionnalité** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Instructions de décalage avec valeur immédiate** | | | | | | | | | | | | | | | | |
| ASR RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 0 | 0 | 1 |
| CSL RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 0 | 1 | 0 |
| CSR RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 0 | 1 | 1 |
| LSL RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 1 | 0 | 0 |
| LSR RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 1 | 0 | 1 |
| ROL RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 1 | 1 | 0 |
| ROR RD,#IMM4 | 0 | 0 | 0 | 0 | 1 | RD | | | IMM4 | | | | 1 | 1 | 1 | 1 |
| **Triadique logique** | | | | | | | | | | | | | | | | |
| AND RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 0 | 0 |
| OR RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 1 | 0 |
| XNOR RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 1 | 1 |

TABLE A.1 – Ce tableau présente le codage binaire des instructions de décalage (shift) et les instruction triadiques (3 registres) sur le *XGate*, comme trouvé dans [19].

Pour les instructions de décalage avec une valeur immédiate, nous définissons alors le sous format `shiftInstructionImm` :

```
7  format shiftInstructionImm #IMM
8    rdIndex := slice{10..8}
9    imm4 := slice{7..4}
10   select slice {3..0}
11     case \b1001 is #ASR
12     case \b1010 is #CSL
13     case \b1011 is #CSR
14     case \b1100 is #LSL
15     case \b1101 is #LSR
16     case \b1110 is #ROL
17     case \b1111 is #ROR
18   end select
19 end format
```

La signature des instructions devient alors `#IMM #ASR` pour l'instruction `ASR`, `#IMM #CSL` pour `CSL`, ... Au niveau des instructions triadiques, la description se fait de la manière suivante :

```
20 format logicalTriadic #Triadic
21   rdIndex := slice{10..8}
22   rs1Index := slice{7..5}
23   rs2Index := slice{4..2}
24   select slice {1..0}
25     case \b00 is #AND
26     case \b10 is #OR
27     case \b11 is #XNOR
28   end select
29 end format
```

Ainsi, nous avons ici décrit le format binaire de ces 10 instructions en 29 lignes.

### A.1.2 Génération du décodeur

Il est possible de générer un décodeur directement, alors que les autres vues (syntaxe et behavior) ne sont pas encore décrites. Ceci permet dans un premier temps de vérifier les erreurs syntaxiques et sémantiques de la description binaire. Il y a notamment une vérification de l'orthogonalité du jeu d'instructions, c'est-à-dire qu'un codage binaire ne peut correspondre qu'à une seule instruction.

### A.1.3 Exploitation des fichiers de sortie

En plus du décodeur (qui se trouve dans les fichiers `instDecoder.h` et `instDecoder.cpp`), un certain nombre de fichiers sont générés pour faciliter la mise au point. Tout d'abord, les fichiers `format_all.dot` et `format_ref.dot` permettent d'afficher l'arbre de description du format des instructions. Le premier affiche l'arbre complet (en indiquant les *format*s ainsi que des instructions de type *select*), voir figure A.1.

Cet arbre devient vite difficile à visualiser, d'où l'importance du deuxième fichier qui permet d'afficher uniquement les étiquettes pour chaque instruction (voir figure A.2). Dans cette dernière figure, il y a maintenant 2 structures arborescentes simples, car les instructions de décalage et les instruction triadiques ne partagent pas de propriétés communes (donc il n'y a pas d'étiquettes communes).

Enfin, en parallèle, un fichier est généré résumant les informations relatives au code binaire des instructions : `instruction.log`. Nous retrouvons alors pour chaque instruction les informations suivantes :

- le parcours de l'arbre permettant de générer l'instruction ;
- la signature de l'instruction : les étiquettes sont classées par ordre alphabétique ;

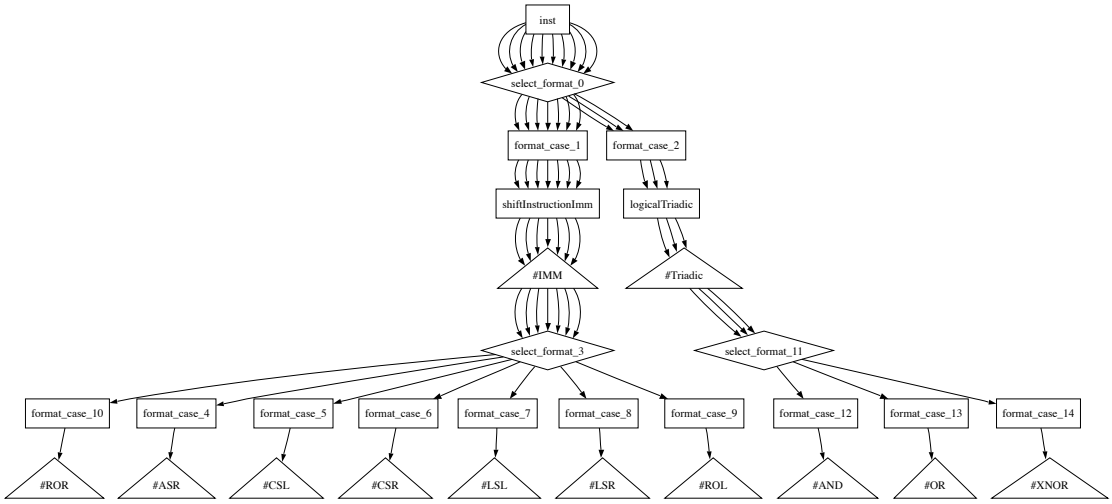



FIGURE A.1 – Arbre généré à partir de la description du code binaire des instructions de décalage (*shift*) et les instruction triadiques (3 registres) du *XGate*.

- le nom utilisé en interne par l'instruction (pour faire de la mise au point au niveau du C++) ;
- le codage binaire, à l'exception des champs de données ;
- les noms des différents champs de données, ainsi que leur taille ;
- le code de l'instruction (en mots dont la taille est définie dans la section *default*) ;

Ainsi, on obtient par exemple pour l'instruction triadique *AND* les informations suivantes :

```
inst
-> select_format_0
-> format_case_2
-> logicalTriadic
-> #Triadic
-> select_format_11
-> format_case_12
-> #AND
instruction id :#AND, #Triadic
Internal name :test_AND_Triadic
Binary coding :00010---------00
data field(s) :rdIndex (3 bits)
               rs1Index (3 bits)
               rs2Index (3 bits)
```

```
Instruction code size :1
```

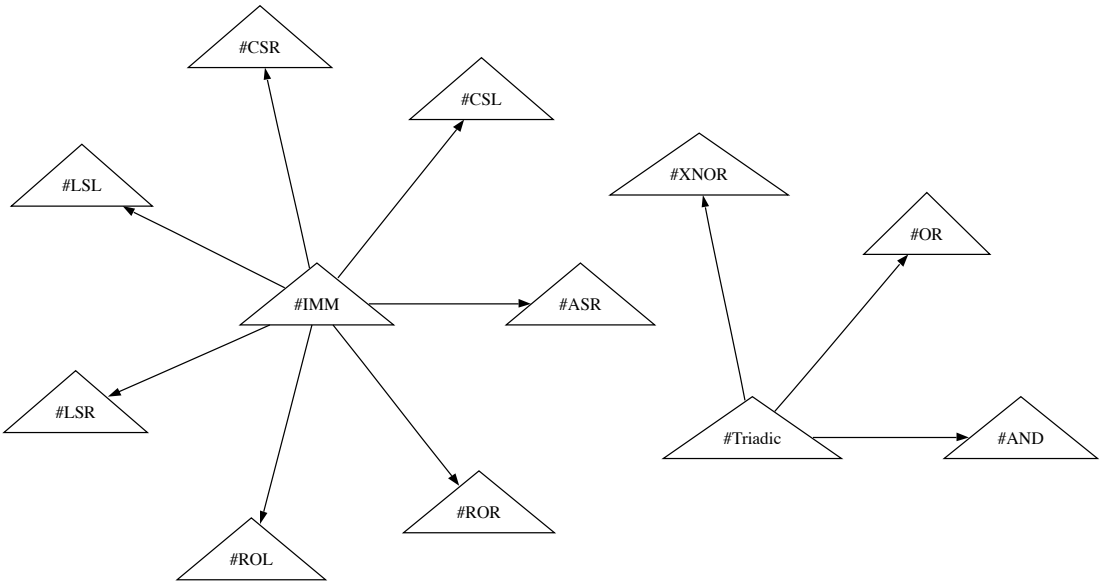



FIGURE A.2 – Arbre généré à partir de la description du code binaire des instructions de décalage (shift) et des instruction triadiques (3 registres) du XGate. Cet arbre est réduit aux étiquettes.

## A.2 Exemple sur la description de la vue syntaxique

Prenons comme exemple l'ensemble d'instructions `triadic` du processeur *XGate*. Dans [19], la description de cet ensemble d'instructions est résumé par le tableau A.2.

| **Fonctionnalité** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Triadique logique** | | | | | | | | | | | | | | | | |
| AND RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 0 | 0 |
| OR RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 1 | 0 |
| XNOR RD,RS1,RS2 | 0 | 0 | 0 | 1 | 0 | RD | | | RS1 | | | RS2 | | | 1 | 1 |
| **Triadique arithmétique** | Utilisé pour comparer : SUB R0, Rs1, Rs2 | | | | | | | | | | | | | | | |
| SUB RD,RS1,RS2 | 0 | 0 | 0 | 1 | 1 | RD | | | RS1 | | | RS2 | | | 0 | 0 |
| SBC RD,RS1,RS2 | 0 | 0 | 0 | 1 | 1 | RD | | | RS1 | | | RS2 | | | 0 | 1 |
| ADD RD,RS1,RS2 | 0 | 0 | 0 | 1 | 1 | RD | | | RS1 | | | RS2 | | | 1 | 0 |
| ADC RD,RS1,RS2 | 0 | 0 | 0 | 1 | 1 | RD | | | RS1 | | | RS2 | | | 1 | 1 |

TABLE A.2 – Ce tableau représente l'ensemble d'instructions `triadic` [19].

Dans HARMLESS , cet ensemble d'instructions peut être décrit comme suit :

```
1  syntax triadicInstruction #TriadicInst
2    field u3 rs1Index
3    field u3 rs2Index
4    field u3 rdIndex
5    triadicOperation
6    "␣R\d,␣R\d,␣R\d", rdIndex, rs1Index, rs2Index
7  end syntax
8
9  syntax triadicOperation
10   select
11     case #AND  "AND"
12     case #XNOR "XNOR"
13     case #SBC  "SBC"
14     case #ADD  "ADD"
15     case #ADC  "ADC"
16   end select
17 end syntax
```

Les syntaxes textuelles des deux instructions `OR` et `SUB` sont décrites à part pour pouvoir décrire syntaxiquement des mnémoniques simplifiés : l'instruction de déplacement `MOV` et l'instruction de comparaison `CMP` respectivement.

```
1  syntax orOperation #TriadicInst #OR
2    field u3 rs1Index
3    field u3 rs2Index
4    field u3 rdIndex
5    if rs1Index = rs2Index then
6      "MOV␣R\d,␣R\d", rdIndex, rs1Index
7    else
8      "OR␣R\d,␣R\d,␣R\d", rdIndex, rs1Index, rs2Index
9    end if;
10 end syntax
11
12 syntax subOperation #TriadicInst #SUB
13   field u3 rs1Index
14   field u3 rs2Index
15   field u3 rdIndex
16   if rdIndex = 0 then
17     "CMP␣R\d,␣R\d",rs1Index, rs2Index
18   else
19     "SUB␣R\d,␣R\d,␣R\d", rdIndex, rs1Index, rs2Index
20   end if
21 end syntax
```

Comme nous l'avons déjà dit, la vue syntaxique est un ensemble d'arbres. L'arbre correspondant à cette description est donné par le schéma A.3.

Comme résultat, pour l'instruction `ADD` par exemple, nous obtenons dans le simulateur le code C++ ci-dessous engendré dans le fichier `instMnemo.cpp`.

```
char* const XGate_ADD_TriadicInst::mnemo()
{
  char *result = new char[64];
  int index = 0;
  index += sprintf (result + index,"ADD");
  index += sprintf (result + index," R\%u, R\%u, R\%u",rdIndex,
                                rs1Index,rs2Index);
  return result;
}
```

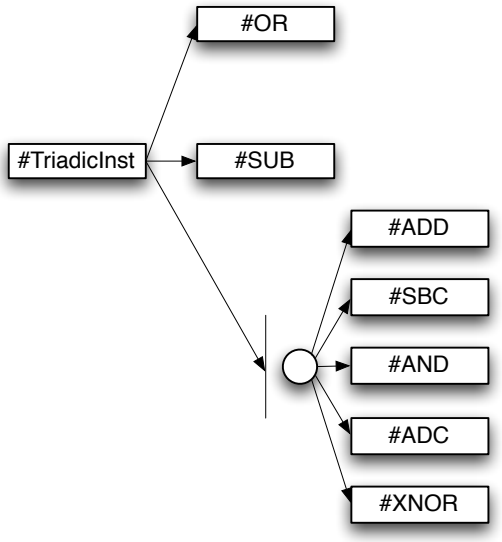



FIGURE A.3 – Syntaxe de l'ensemble d'instructions triadic.

## A.3 Exemple sur la description de la vue sémantique

De même, prenons comme exemple l'ensemble d'instructions `triadic` du XGate [19] décrit par le tableau A.2.

Dans HARMLESS, cet ensemble d'instructions peut être décrit comme suit :

```
1  behavior triadicInstruction() #TriadicInst
2    field u3 rs1Index;
3    field u3 rs2Index;
4    field u3 rdIndex;
5    u16 rs1Value;
6    u16 rs2Value;
7    u16 rdValue;
8    do
9      rs1Value := GPR.read16(rs1Index);
10     rs2Value := GPR.read16(rs2Index);
11   end do
12   select
13     case #AND  do rdValue := ALU.andOpUpdateCCR(rs1Value,
           rs2Value);end do
14     case #OR   do rdValue := ALU.orOpUpdateCCR(rs1Value,
           rs2Value); end do
15     case #XNOR do rdValue := ALU.xnorOpUpdateCCR(rs1Value,
           rs2Value);end do
16     case #SUB  do rdValue := ALU.sub16OpUpdateCCR(rs1Value,
           rs2Value);end do
17     case #SBC  do rdValue := ALU.sub16OpWithCarryUpdateCCR(
           rs1Value, rs2Value);end do
18     case #ADD  do rdValue := ALU.add16OpUpdateCCR(rs1Value,
           rs2Value);end do
19     case #ADC  do rdValue := ALU.add16OpWithCarryUpdateCCR(
           rs1Value, rs2Value);end do
20   end select
21   do GPR.write16(rdIndex,rdValue); end do
22 end behavior
```

Comme nous avons déjà dit, la vue sémantique est un ensemble d'arbres. L'arbre correspondant à cette description est donné par le schéma A.4.

En comparant les deux schémas (A.3 et A.4), nous remarquons que les deux arbres ne sont pas exactement les mêmes, cela montre la caractéristique de notre langage qui permet à l'utilisateur de choisir la façon la plus adéquate pour décrire une vue.

Comme résultat, pour l'instruction `ADD` par exemple, nous obtenons dans le simulateur, le code C++ ci-dessous engendré dans le fichier `instExec.cpp`.

```
void XGate_ADD_TriadicInst::exec(arch *_arch){
```

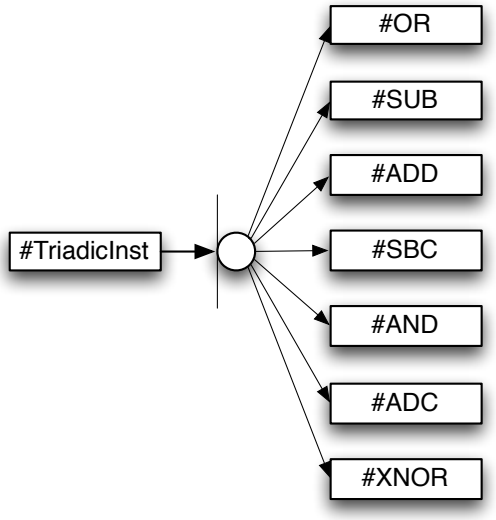



FIGURE A.4 – Sémantique de l'ensemble d'instructions triadic.

```
  u16 triadicInstruction_rs1Value; //u16
  u16 triadicInstruction_rs2Value; //u16
  u16 triadicInstruction_rdValue; //u16
  triadicInstruction_rs1Value = (_arch->GPR_read16(this->rs1Index))
                                & 0xFFFF;
  triadicInstruction_rs2Value = (_arch->GPR_read16(this->rs2Index))
                                & 0xFFFF;
  triadicInstruction_rdValue = (_arch->ALU_add16OpUpdateCCR(
        triadicInstruction_rs1Value, triadicInstruction_rs2Value))
                                & 0xFFFF;
  _arch->GPR_write16(this->rdIndex, triadicInstruction_rdValue);
}
```

## A.4 Exemple sur la modélisation des instructions

Nous avons déjà dit que chaque instruction dans le simulateur engendré est modélisée comme une classe *C++*. Pour illustrer ça, prenons comme exemple l'instruction d'addition tirée de l'ensemble d'instructions `triadic` du XGate [19] décrit par le tableau A.2. La classe qui modélise cette instruction est la suivante :

```
class XGate_ADD_TriadicInst : public XGate_instruction{
  u8 rs1Index;  /* registre source */
  u8 rs2Index;  /* registre source */
  u8 rdIndex;  /* registre destination */

  public:
    /* le constructeur */
    XGate_ADD_TriadicInst(const u16 pc, const u16 chunk1);
    virtual ~XGate_ADD_TriadicInst() {};  /* le destructeur */
    virtual void exec(arch *);  /* la fonction d'execution */
    /* la fonction mnémonique */
    virtual const char* const mnemo(const u32);
    ...
};
```

Les trois méthodes principales qui composent cette classe sont :

**1. Le constructeur :**

```
XGate_ADD_TriadicInst::XGate_ADD_TriadicInst(const u16 pc, const u16
      chunk1) : XGate_instruction(pc)
{
  // constructeur de rs1Index (u3)
  rs1Index = FIELD(chunk1,(7UL),(5UL));
  // constructeur de rs2Index (u3)
  rs2Index = FIELD(chunk1,(4UL),(2UL));
  // constructeur de rdIndex (u3)
  rdIndex = FIELD(chunk1,(10UL),(8UL));
}
```

**2. La fonction d'exécution :**

```
  void XGate_ADD_TriadicInst::exec(arch *_arch){
  u16 triadicInstruction_rs1Value; //u16
  u16 triadicInstruction_rs2Value; //u16
  u16 triadicInstruction_rdValue; //u16
  triadicInstruction_rs1Value = (_arch->GPR_read16(this->rs1Index))
                                & 0xFFFF;
  triadicInstruction_rs2Value = (_arch->GPR_read16(this->rs2Index))
                                & 0xFFFF;
  triadicInstruction_rdValue = (_arch->ALU_add16OpUpdateCCR(
       triadicInstruction_rs1Value, triadicInstruction_rs2Value))
                      & 0xFFFF;
```

```
  _arch->GPR_write16(this->rdIndex, triadicInstruction_rdValue);

}
```

**3. La fonction mnémonique :**

```
const char* const XGate_ADD_TriadicInst::mnemo(const u32
       instructionAddress)
{
  char *result = new char[64];
  int index = 0;
  index += sprintf (result + index,"ADD");
  index += sprintf (result + index," R\%u, R\%u, R\%u",(u32)(rdIndex),
                         (u32)(rs1Index),(u32)(rs2Index));
  return result;
}
```

# Annexe B

# Les aléas dans un pipeline

## B.1 Les aléas structurels

Dans un pipeline idéal, nous supposons que les étages du pipeline sont fonctionnellement indépendants. En réalité, ce n'est pas tout à fait vrai. En effet, un pipeline peut avoir un nombre important d'étages, par contre les ressources matérielles (comme les registres, l'accès mémoire, l'unité logique et arithmétique,...) seront toujours en nombre limité. Ce manque de ressources peut causer des aléas structurels.

Par exemple, il peut arriver que deux instructions dans des étages différents du pipeline veulent utiliser une même unité fonctionnelle simultanément. Dans ce cas, la seconde instruction doit être suspendue tant que la première ne libère pas la ressource en question. Pour éviter ce genre d'aléas, la duplication des ressources est généralement utilisée, ce qui augmente la quantité de matériel dans le processeur. Sur la figure B.1, un aléa structurel est levé. En effet, supposons que l'instruction *i-2* accède à la mémoire en écriture dans l'étage $E$ et que nous disposons d'une mémoire autorisant un seul accès (il n'est pas possible de faire un accès lecture et écriture simultanément). À l'instant $t$, le pipeline ne laisse pas l'instruction $i$ de rentrer dans l'étage $F$ (accès à la mémoire en lecture pour lire cette instruction), puisque l'accès à la mémoire n'est pas autorisé. Une bulle est insérée à l'instant $t$ et le début d'exécution de l'instruction $i$ est décalé d'un cycle d'horloge. Dans ce cas simple, un recours à une mémoires autorisant deux accès simultanés permet de découpler les deux accès et ainsi donne la possibilité à l'instruction $i$ de finir son exécution à l'instant *t+4*.

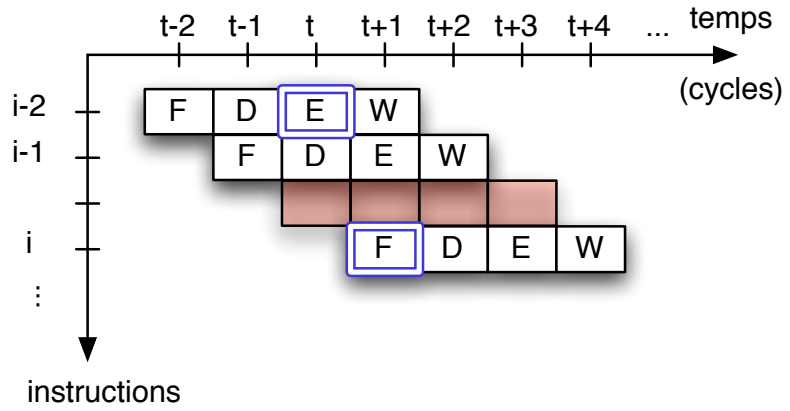



FIGURE B.1 – L'aléa structurel conduit à l'insertion des bulles dans un pipeline. L'instruction *i* aurait due normalement sortir du pipeline à l'instant *t+3*, mais à cause de l'aléas structurel avec l'instruction *i-2*, elle a fini un cycle après.

## B.2 Les aléas de données

Comme nous l'avons déjà précisé, les aléas de données interviennent, dans le cas où une instruction essaie d'accéder à un registre et que ce même registre est utilisé par une instruction précédente, mais qui est encore dans le pipeline.

Plusieurs types de dépendances entre les données peuvent survenir (voir figure B.2). Soit deux instructions $i$ et $j$ consécutives :

**LAE (lecture après écriture), ou *RAW* (*Read After Write*) :**
C'est le cas le plus rencontré et qui arrive quand l'instruction $j$ essaie de lire une donnée avant que l'instruction $i$ ne l'ai écrite. (Exemple : instruction 1 et 2 de la figure B.2) ;

**EAE (écriture après écriture), ou *WAW* (*Write After Write*) :**
Dans ce cas, l'instruction $j$ essaie d'écrire une donnée dans un registre avant que l'instruction $i$ ne l'y ait écrite. Les écritures se font dans le mauvais ordre et ainsi les résultats seront erronés. Cet aléa peut se produire surtout dans le cas où plus d'un étage du pipeline autorise l'écriture [1]. (Exemple : instruction 3 et 4 de la figure B.2) ;

**EAL (écriture après lecture), ou *WAR* pour (*Write After Read*) :**
L'instruction $j$ essaie d'écrire dans un registre avant que l'instruction $i$ n'y

1. Ou en cas d'exécution dans le désordre dans un processeur superscalaire à ordonnancement dynamique.

ait lu sa donnée. Ce cas de figure ne se présente que lorsqu'un pipeline autorise l'écriture des résultats dans un étage situé avant de celui permettant la lecture des données. (Exemple : instruction 2 et 3 de la figure B.2).

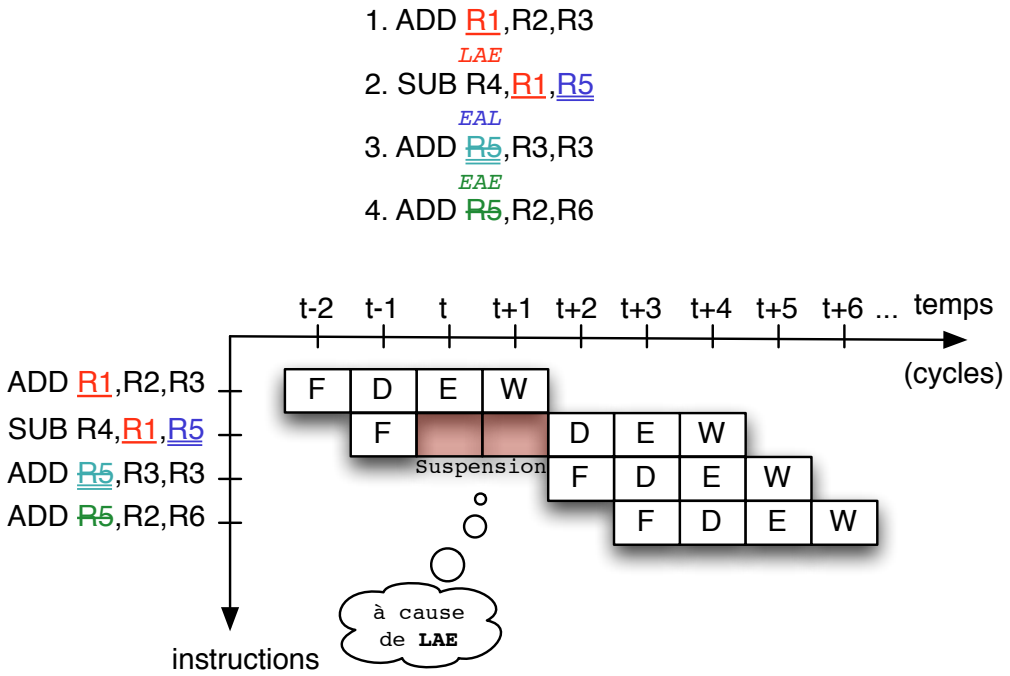



FIGURE B.2 – Les aléas de données retardent l'exécution des instructions dans un pipeline. Ici, le pipeline à 4 étages permet aux instructions de lire les opérandes dans l'étage $F$ et écrire le résultat dans l'étage $W$.

Plusieurs méthodes sont utilisées pour réduire l'influence des aléas de données dans un pipeline. À savoir, le renommage de registres, les raccourcis matériels (ou court-circuits ou envoi), une optimisation du compilateur,... [29].

## B.3 Les aléas de contrôle

Les instructions de branchement, très présentes ($\simeq$ 15%, [29]) dans les programmes classiques (benchmarks), sont la cause des aléas de contrôle. Elles peuvent ralentir l'exécution des instructions et par la suite dégrader la performance du processeur.

En effet, lorsqu'un branchement est pris (c'est-à-dire branchement inconditionnel, ou dans le cas où la condition est vrai si le branchement est conditionnel.

Dans tout le cas, l'instruction en question provoque la modification du *PC*), les instructions suivantes qui ont commencé leur exécution doivent être annulées (suspension du pipeline, en insérant des bulles), et l'instruction cible du branchement doit entamer son exécution comme le montre la figure B.3. Ici, l'instruction de branchement *i-1* calcule sa cible pendant l'étage *D*. Donc à l'instant *t*, et comme nous ne savons pas encore si le branchement sera pris ou non, l'instruction entrante dans le pipeline est celle qui suit directement l'instruction de branchement *i-1*. En calculant, l'adresse de l'instruction cible à *t+1* (branchement pris), l'instruction *i* est annulée et l'instruction *i+5* commence son exécution, ce que nous fait perdre un cycle d'horloge. En général, le délai de branchement en nombre de cycles d'horloge augmente avec la profondeur du pipeline. Pour augmenter la vitesse des processeurs, les pipelines sont de plus en plus longs pour diminuer la quantité de travail réalisée dans chaque étage. Par exemple, l'Intel *Pentium IV* contient un pipeline à 20 étages, soit le double que son prédécesseur le *Pentium III*.

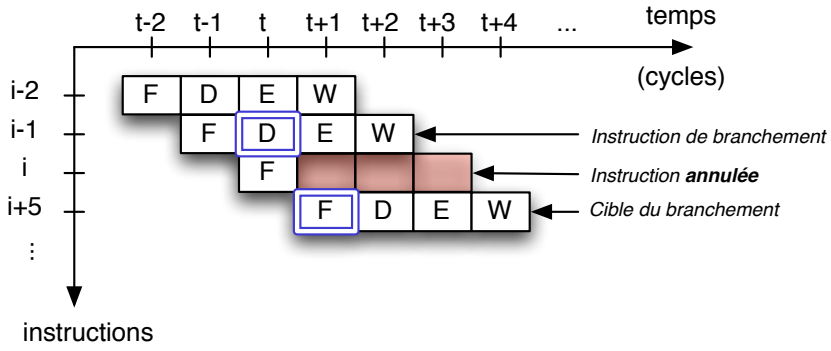



FIGURE B.3 – Les aléas de contrôle retardent l'exécution des instructions dans un pipeline. L'instruction *i* a arrêté son exécution après que l'instruction de branchement *i-1* a modifié l'adresse du *PC* pour y mettre l'adresse destination.

Il existe plusieurs techniques pour essayer de diminuer les suspensions de pipeline causées par les aléas de contrôle, et ceci en détectant au plus tôt les instructions de branchement dans le pipeline, comme les algorithmes de prédiction de branchement statique ou dynamique, l'exécution spéculative des instructions, le déroulage de boucles qui permet de réduire la fréquence de branchements de boucle.

# Annexe C

# Exemple de description d'un pipeline

Cet exemple est basé sur un pipeline de 6 étages présenté dans le chapitre 5, section 5.4.2

La description de ce pipeline ainsi que des ressources internes et externes, les classes d'instructions et les classes d'instructions de dépendance de données se fait de la manière suivante :

```
1: version 7

2: model XGate

3: pipeline pipe
4: pipe addStage Fetch 1
5: pipe addStage Decode 1
6: pipe addStage Execute1 1
7: pipe addStage Execute2 1
8: pipe addStage Memory 1
9: pipe addStage Register 1

10: pipe addResource external DataDep1 0 priority
11: pipe addResource external DataDep2 0 priority
12: pipe addResource external loadStore 0 priority
13: pipe addResource external fetch 0 priority
14: pipe addResource internal 1 alu 0 priority

15: pipe addInstructionClass C1
16: C1 dependResource 1 DataDep1 Decode
```

```
17: C1 dependResource 1 loadStore Memory
18: C1 dependResource 1 alu Execute1
19: C1 getResource 1 alu Execute1
20: C1 releaseResource 1 alu Execute2
21: C1 dependResource 1 fetch Fetch

22: pipe addInstructionClass C2
23: C2 dependResource 1 DataDep1 Decode
24: C2 dependResource 1 DataDep2 Memory
25: C2 dependResource 1 loadStore Memory
26: C2 dependResource 1 alu Execute1
27: C2 getResource 1 alu Execute1
28: C2 releaseResource 1 alu Execute2
29: C2 dependResource 1 fetch Fetch

30: pipe addInstructionClass C3
31: C3 dependResource 1 fetch Fetch

32: pipe addInstructionClass C4
33: C4 dependResource 1 DataDep1 Decode
34: C4 dependResource 1 fetch Fetch

35: pipe addInstructionClass C5
36: C5 dependResource 1 DataDep1 Decode
37: C5 dependResource 1 alu Execute1
38: C5 getResource 1 alu Execute1
39: C5 releaseResource 1 alu Execute2
40: C5 dependResource 1 fetch Fetch

41: pipe addInstructionClass C6
42: C6 dependResource 1 alu Execute1
43: C6 getResource 1 alu Execute1
44: C6 releaseResource 1 alu Execute2
45: C6 dependResource 1 fetch Fetch

46: pipe addDataDependencyInstructionClass  ddC1
47: ddC1 required Fetch
48: ddC1 lock Decode
49: ddC1 unlock Register
```

```
50: pipe addDataDependencyInstructionClass  ddC2
51: ddC2 lock Decode
52: ddC2 unlock Register

53: pipe addDataDependencyInstructionClass  ddC3
54: ddC3 required Fetch

55: pipe addDataDependencyInstructionClass  ddC4
56: ddC4 required Fetch
57: ddC4 required Execute2

58: pipe addDataDependencyInstructionClass  ddC5
59: ddC5 required Fetch
60: ddC5 lock Decode
61: ddC5 required Execute2
62: ddC5 unlock Register
```

La ligne 1 indique la version de l'outil *p2a*. La ligne 2 donne le nom du modèle de processeur (ici, le co-processeur *XGate* de chez *Freescale*). La ligne 3 déclare un pipeline nommé `pipe`. Les lignes 4 à 9 déclarent les 6 étages du pipeline (les noms des étages sont les suivants : `Fetch`, `Decode`, `Execute1`, `Execute2`, `Memory` et `Register`). Ensuite les lignes 10 à 14 donnent les différentes ressources internes et externes ainsi que leur priorité. De la ligne 15 à 45, les classes d'instructions sont définies. Par exemple, la classe d'instructions `C4` dépend de la ressource `dataDep1` pour entrer dans l'étage `Decode` (le chiffre 1 indique que chaque instruction de cette classe utilise la ressource en question une seule fois dans l'étage `Decode`). Enfin, les lignes 46 à 62 définissent les classes d'instructions de dépendance de données. Par exemple, la classe `ddC1` lit ses opérandes dans l'étage `Fetch` et écrit les résultats dans l'étage `Register` (donc il faut bloquer dans l'étage `Decode` les registres qui seront mis à jour dans l'étage `Register` où ils seront débloqués [1], pour que les instructions qui vont lire dans ces registres ne puissent pas avancer dans le pipeline avant que leurs opérandes ne soient pas disponibles).

1. Dans le cas où il y a un raccourci matériel (appelés également court-circuit ou envoi), permettant d'envoyer le résultat d'une opération aux instructions le nécessitant dans l'étage `Execute2`, les registres seront débloqués dans l'étage `Execute2`.

# Publications liées à la thèse

1. Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet and Guillaume Savaton. Simulator Generation Using an Automaton Based Pipeline Model for Timing Analysis. *In International Multiconference on Computer Science and Information Technology (IMCSIT), International Workshop on Real Time Software (RTS'08)*, Wisla, Poland , 2008. IEEE.

2. Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet and Guillaume Savaton. Instruction set simulator generation using HARMLESS, a new hardware architecture description language. *In Simutools '09 : Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Rome, Italy, 2009. ICST and ACM (SIGSIM and SIGMETRICS).

3. Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet and Guillaume Savaton. Cycle Accurate Simulator Generation Using HARMLESS. *In international Middle Eastern Multiconference on Simulation and Modelling (MESM'09)*, Beirut, Lebanon, September 2009. EUROSIS and IEEE UKRI - SPC.

# Bibliographie


[1] Chess/checkers : a retargetable tool-suite for embedded processors. Technical white paper, *http ://www.retarget.com*, June 2003.

[2] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6) :509–516, 1978.

[3] Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. In *MICRO 28 : Proceedings of the 28th annual international symposium on Microarchitecture*, pages 46–56, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[4] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, and Dietmar Voggenauer. The mimola language version 4.1. Technical report, Lehrstuhl Informatik XII University of Dortmund, Dortmund, 1994.

[5] Michael K. Becker, Michael S. Allen, Charles R. Moore, John S. Muhich, and David P. Tuttle. The power pc 601 microprocessor. *IEEE Micro*, 13(5) :54–68, 1993.

[6] Mikaël Briday. *Validation par simulation fine d'une architecture opérationnelle.* Thèse de doctorat de l'Université de Nantes, 2004.

[7] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Modelling of superscala processor architectures with design/CPN. In K. Jensen, editor, *Daimi PB-532 : Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 15–30. Aarhus University, June 1998.

[8] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Wcet analysis of superscalar processors using simulationwith coloured petri nets. *Real-Time Syst.*, pages 275–288, 2000.

[9] Giorgio Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications.* Springer Science+Business Media, Inc., 2nd edition, 2005.

[10] Philippe Schnoebelen (coordinateur). *Vérification de logiciel : Techniques et outils du model-checking - Livre collectif.* Vuibert, Avril 1999.

[11] CoWare, LISATek Product Family. *LISA Language Reference Manual*, 2004.

[12] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel. Effective control for pipelined computers. In *Spring COMPCON75 Digest of Papers*, pages 181–184, February 1975.

[13] Anne-Marie Déplanche and Francis Cottet. *Section « Ordonnancement temps réel ». Encyclopédie de l'informatique et des systèmes d'information.* Vuibert, Paris, 2006.

[14] M. Anton Ertl and Andreas Krall. Instruction scheduling for complex pipelines. In *CC '92 : Proceedings of the 4th International Conference on Compiler Construction*, pages 207–218, London, UK, 1992. Springer-Verlag.

[15] Sébastien Faucou. *Description et construction d'architectures opérationnelles validées temporellement.* Thèse de doctorat de l'Université de Nantes, 2002.

[16] A. Fauth and A. Knoll. Automated generation of dsp program development tools using a machine description formalism. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 457–460. IEEE, 1993.

[17] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95 : Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[18] M. Freericks. The nml machine description formalism. Technical Report 1991/15, Computer science department, TU Berlin, Germany, 1991.

[19] Freescale Semiconductor, Inc. *XGATE Block Guide*, 2003.

[20] Freescale Semiconductor, Inc. *MPC750 RISC Microprocessor Family User's Manual*, 2004.

[21] Freescale Semiconductor, Inc. *e200z1 Power Architecture Core Reference Manual*, 2008.

[22] Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. Rtgen : An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS '99 : Proceedings of the 12th international symposium on System synthesis*, page 44, Washington, DC, USA, 1999. IEEE Computer Society.

[23] Peter Grun, Ashok Halambi, Nikil D. Dutt, and Alexandru Nicolau. Rtgen-an algorithm for automatic generation of reservation tables from architectural descriptions. *IEEE Trans. VLSI Syst.*, 11(4) :731–737, 2003.

[24] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl : an instruction set description language for retargetability. In *DAC '97 : Proceedings of*

*the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM.

[25] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *DAC '99 : Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 927–932, New York, NY, USA, 1999. ACM.

[26] A. Halambi, P. Grun, and al. Expression : A language for architecture exploration through compiler/simulator retargetability. In *European Conference on Design, Automation and Test (DATE)*, March 1999.

[27] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In *DAC '98 : Proceedings of the 35th annual Design Automation Conference*, pages 510–515, New York, NY, USA, 1998. ACM.

[28] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach-Second Edition*. Morgan Kaufmann Publishers, Inc., 2001.

[30] ISO and IEC, editors. *Information technology - Syntactic metalanguage - Extended BNF*. ISO/IEC 14977 : 1996(E), 1996.

[31] K. Jensen. *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use*. New York : Springer-Verlag, 1997.

[32] Asheesh Khare, Ashok Halambi, Nicolae Savoiu, Peter Grun, Nikil Dutt, and Alex Nicolau. V-sat : A visual specification and analysis tool for system-on-chip exploration. *Journal of Systems Architecture*, 47(3-4) :263 – 275, 2001.

[33] Kevin Kranen. *`SystemC` 2.0.1 User's Guide*. Synopsys, Inc.

[34] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1) :6–22, 1984.

[35] J.Y.T. Leung, editor. *Handbook of Scheduling*. Chapman & Hall, CRC Press, 2004.

[36] F. Lohr, A. Fauth, and M. Freericks. Sigh/sim – an environment fo retargetable instruction set simulation. Technical Report 1993/43, Computer science department, TU Berlin, Germany, 1993.

[37] O. Lüthje. A methodology for automated test generation for lisa processor models. In *In Proceedings of Synthesis and System Integration of Mixed Technologies (SASIMI)*, pages 266–273, 2004.

[38] Scott McFarling. Combining branch predictors. Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA, 1993.

[39] Prabhat Mishra and Nikil Dutt. Architecture description languages for programmable embedded systems. In *IEE Proceedings on Computers and Digital Techniques*, pages 285 – 297, 2005.

[40] Prabhat Mishra and Nikil Dutt, editors. *Processor description languages*. Morgan Kaufmann Publishers, 2008.

[41] Thomas Müller. Employing finite automata for resource scheduling. In *MICRO 26 : Proceedings of the 26th annual international symposium on Microarchitecture*, pages 12–20, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[42] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02 : Proceedings of the 39th annual Design Automation Conference*, pages 22–27, New York, NY, USA, 2002. ACM.

[43] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2) :196–255, 1995.

[44] Stéphane Pailler. *Analyse Hors Ligne d'Ordonnançabilité d'Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques*. Thèse de doctorat de l'Université de Poitiers, 2006.

[45] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4) :815–834, 2000.

[46] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99 : Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 933–938, New York, NY, USA, 1999. ACM.

[47] Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *POPL '94 : Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 280–286, New York, NY, USA, 1994. ACM Press.

[48] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *VLSID '99 : Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, page 132, Washington, DC, USA, 1999. IEEE Computer Society.

[49] Tahiry Ratsiambahotra, Hugues Cassé, and Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. In *SPECTS'09 : Proceedings of the 12th international conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems*, pages 65–72, Piscataway, NJ, USA, 2009. IEEE Press.

[50] Tahiry Ratsiambahotra, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Génération automatique de simulateurs fonctionnels de processeurs. In *Symposium sur les Architectures Nouvelles de Machines (SympA), Fribourg, 11/02/2008-13/02/2008*, page (support électronique), http ://www.eif.ch, février 2008. Ecole d'ingénieurs et d'architectes de Fribourg.

[51] Rami R. Razouk. The use of petri nets for modeling pipelined processors. In *DAC '88 : Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 548–553, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[52] Mehrdad Reshadi, Bita Gorjiara, and Nikil D. Dutt. Generic processor modeling for automatically generating very fast cycle-accurate simulators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(12) :2904–2918, Dec. 2006.

[53] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation : a technique for fast and flexible instruction set simulation. In *DAC '03 : Proceedings of the 40th annual Design Automation Conference*, pages 758–763, New York, NY, USA, 2003. ACM.

[54] Vincent-Xavier Reynaud. Développement d'un outil d'opérations sur les automates dans le cadre de la réalisation d'un hadl. Master's thesis, Institut de Recherche en Communication et Cybernétique de Nantes, 2007.

[55] Oliver Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, Mario Steinert, Gunnar Braun, and Achim Nohl. Rtl processor synthesis for architecture exploration and implementation. In *DATE '04 : Proceedings of the conference on Design, automation and test in Europe*, page 30156, Washington, DC, USA, 2004. IEEE Computer Society.

[56] James E. Smith. A study of branch prediction strategies. In *ISCA '81 : Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[57] John A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, 1988.

[58] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems ? *Real-Time Syst.*, 2(4) :247–254, 1990.

[59] Yvon Trinquet and al. *Section « Systèmes temps réel ». Encyclopédie de l'informatique et des systèmes d'information.* Vuibert, Paris, 2006.

[60] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *MICRO 24 : Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, New York, NY, USA, 1991. ACM.

[61] G. Zimmermann. The mimola design system a computer aided digital processor design method. In *DAC '79 : Proceedings of the 16th Design Automation Conference*, pages 53–58, Piscataway, NJ, USA, 1979. IEEE Press.

[62] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design, 1996.