Synthèse automatisée de circuits asynchrones optimisés prouvés quasi insensibles aux délais

V. Brégier

Institut National Polytechnique de Grenoble

N° attribué par la bibliothèque
978-2-84813-105-4

THÈSE

pour obtenir le grade de

Docteur de l'INP Grenoble

Spécialité : **Micro et Nano Électronique**

préparée au laboratoire **TIMA – Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs**

dans le cadre de l'École Doctorale **EEATS – Électronique, Électrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement
par

Vivian BRÉGIER

le 14 Septembre 2007

Titre:

Synthèse automatisée de circuits asynchrones optimisés prouvés Quasi Insensibles aux Délais

Directeur de thèse: **Laurent Fesquet**
Co-directeur de thèse: **Marc Renaudin**

Jury

M. Frédéric Pétrot,	Président du jury
M. Habib Mehrez,	Rapporteur
M. Patrice Quinton,	Rapporteur
M. Jean-Baptiste Rigaud,	Examinateur
M. Laurent Fesquet,	Directeur de thèse
M. Marc Renaudin,	Co-directeur de thèse

Résumé

Dans un circuit asynchrone, la synchronisation entre les blocs est locale : on s'affranchit ainsi des contraintes liées à l'horloge. Ces circuits sont plus robustes, modulaires, moins bruités, et ont une consommation dynamique plus faible que les circuits synchrones. Cependant, le manque d'outils de conception de tels circuits freine leur développement. Cette thèse a permi de développer une technique de synthèse automatique de circuits asynchrones quasi insensibles aux délais (QDI), qui sont particulièrement robustes. La méthode de synthèse permet de synthétiser un circuit totalement décomposé en portes logiques élémentaires, ce qui permet d'effectuer une projection technologique. De plus, une étude formelle réalisée durant la thèse démontre que les circuits synthétisés respectent la contrainte de quasi insensibilité aux délais. Cette technique de synthèse a été développé au sein du projet TAST. Elle a été validée sur un ensemble de circuits de tests.

Abstract

In an asynchronous circuit, the synchronization between the blocs is local : the constraints due to the clock do not apply. These circuits are more robust, modular, have less noise and a lower dynamic power consumption that asynchronous circuits. However, the lack of design tools for such circuits prevents them from spreading widely. This thesis aimed at developping an automatic synthesis technique targeting asynchronous quasi delay insensitive (QDI) circuits, which are particularly robust. The technique synthesizes a circuit totally decomposed in elemetary logical gates, which allows a later technology mapping. Moreover, a formal study done during this thesis proves that the circuits synthesized respect the constraint of quasi delay insensitivity. This synthesis technique was developped in the TAST project. Is has been validated on a set of test circuits.

Remerciements

Pour commencer, je voudrais remercier Marc Renaudin, qui m'a acceuilli dans son groupe de recherche, le Groupe CIS, et m'a fait découvrir le monde des circuits asynchrones. J'ai appris beaucoup de choses à son contact, et j'ai énormément apprécié travailler avec quelqu'un qui ait à la fois le recul nécessaire pour l'encadrement d'un travail de thèse, mais aussi une grande compétence sur le plan technique.

Je souhaite aussi remercier Laurent Fesquet pour son encadrement et son aide, en particulier pour sa relecture rapide et efficace de ce manuscrit.

Tout naturellement, je remercie mes rapporteurs, Patrice Quinton et Habib Mehrez, pour le temps qu'ils ont consacré à lire attentivement et dans le détail mes travaux de thèse, ainsi pour les améliorations du manuscrit dont ils ont été l'origine.

Je remercie également Frédéric Pétrot, qui a présidé le jury de ma thèse, et qui a lui aussi été la source de nombreuses améliorations de ce manuscrit, ainsi que Jean-Baptiste Rigaud, qui a participé au jury.

Merci à Yann Remond, Bertrand Folco, Yannick Monnet, Joao Fragoso, avec qui j'ai beaucoup collaboré. J'ai beaucoup aimé travailler avec eux.

Merci aussi David Rios pour avoir eu confiance en moi au moment d'un certain pari. L'histoire lui a donné raison.

Plus généralement, un grand merci à tout le groupe CIS, pour l'ambiance détendue, chaleureuse et amicale qui y règne. Je ne vais pas citer tout le monde, ils sont trop nombreux.

Un grand merci à tous mes amis dixsousiens, qui se reconnaitront. Peut-être que ma productivité à cour-terme aurait été meilleure sans le canal irc que j'ai fréquenté assidument, mais ma motivation, elle, aurait vite chuté. Merci pour vos conseils et votre soutient.

Enfin, je remercie mes parents, sans qui rien de tout cela n'aurait pu arriver

Pour finir, je remercie Marion, ma compagne, bientôt ma femme, pour son soutien, et surtout pour tout ce qu'elle apporte dans ma vie.

Table des matières

Résumé . . . iii
Abstract . . . iv
Remerciements . . . v
Table des matières . . . vii

Introduction **1**

1 Les circuits asynchrones Quasi Insensibles aux Délais **7**
1 Les circuits asynchrones . . . 7
1.1 Intérêt des circuits sans horloge . . . 8
1.2 Synchronisation locale et canaux de communication . . . 8
1.3 Protocole poignée de mains . . . 9
2 Classes de circuits asynchrones . . . 11
2.1 Circuits Insensibles aux Délais . . . 11
2.2 Circuits Quasi Insensibles aux Délais . . . 12
2.3 Circuits Indépendants de la Vitesse . . . 13
2.4 Circuits Micropipeline . . . 13
2.5 Circuits de Huffman . . . 14
3 Quasi insensibilité aux délais . . . 14
3.1 Intérêt de l'insensibilité aux délais . . . 14
3.2 Fourche isochrone et Turing-complétude . . . 15
3.3 Aléas . . . 15
3.4 Codage des données insensible aux délais . . . 15
3.5 Mode *push/pull* . . . 16
3.6 Reset . . . 17
3.7 Initialisation . . . 17
3.8 Vérification de la quasi insensibilité aux délais . . . 18
3.9 Protocoles . . . 19
4 Conclusion . . . 23

2 Techniques de synthèse de circuits asynchrones **25**
1 Introduction . . . 25
2 Synthèse dirigée par la syntaxe . . . 25
2.1 TIDE (Handshake Solutions) . . . 26
2.2 Balsa . . . 27
3 Synthèse par compilation . . . 28
3.1 CAST . . . 28
3.2 TAST (ancienne version) . . . 29
4 Autres méthodes . . . 31

4.1 Petrify 31
4.2 minimalist 32
4.3 NCL : Null Convention Logic 32

3 Modélisation de circuits QDI **35**
1 Introduction 35
2 Modèles comportementaux 36
2.1 Comportement d'un circuit QDI 36
2.2 Processus communicants : Le langage CHP 37
2.3 Raffinement DTL 39
2.4 Modèle de Diagramme de Décision Multi-valué (MDD) 41
3 Modèles structurels 46
3.1 Structure d'un circuit QDI 46
3.2 Netlist 47
3.3 Hiérarchie 49
3.4 Règles de productions 51
4 Conclusion 52

4 Synthèse et optimisation de circuits QDI **53**
1 Introduction 53
2 Génération du MDD 53
3 Synthèse basique du chemin de données : DIMS 54
3.1 Présentation de la synthèse DIMS 54
3.2 Algorithme de synthèse DIMS à partir d'un MDD 55
4 Optimisations du MDD 56
4.1 Nécessité de l'optimisation 56
4.2 Algorithmes d'optimisation 57
5 Limitation du fan-in : synthèse en portes à deux entrées 63
5.1 Intérêt de la limitation du fan-in 63
5.2 Algorithme de synthèse de MDD en portes à deux entrées . . 66
6 Probabilités et équilibrage des arbres de portes OR 68
6.1 Motivation 68
6.2 Probabilité de transition 69
6.3 Calcul des probabilités à partir du MDD 70
6.4 Algorithme de génération de l'arbre optimisé 72
7 Conclusion 73

5 Correction formelle du flot de synthèse **75**
1 Introduction 75
2 Caractérisation formelle de la quasi insensibilité aux délais 75
3 La synthèse DIMS génère des circuits QDI 76
3.1 Génération des règles de production à partir du MDD 76
3.2 Conditions sur le MDD 78
3.3 Non-interférence 79
3.4 Stabilité 79
4 La synthèse en porte 2 entrées génère des circuits QDI 82
4.1 Règles de production 82
4.2 Similitude avec la synthèse DIMS 83

4.3 Non-interférence 83
4.4 Stabilité 83
5 Étapes de transformation 85
5.1 Optimisations du MDD 85
5.2 Projection technologique : fusion de portes 86
6 Conclusion 88

6 Implémentation et Résultats **89**
1 Présentation de TAST 89
2 Circuits d'évaluation 91
2.1 Principe 91
2.2 Composants de base 92
2.3 Présentation des circuits 94
3 Résultats 96
4 Conclusion 96

Conclusion **99**

Bibliographie **101**

Introduction

Contexte

Depuis les débuts de la microélectronique, l'industrie a choisi la solution de la discrétisation du temps pour résoudre les problèmes de synchronisation au sein d'une puce : ce sont les circuits synchrones.

Cette solution simplifie grandement la conception et le test du circuit, puisqu'un intervalle de temps discrétisé ne contient qu'un nombre fini et connu d'instants. Pour implémenter la discrétisation du temps, et synchroniser tous les composants du circuit, un signal global est nécessaire : on l'appelle signal d'horloge.

Cependant, la discrétisation du temps pose de nombreux problèmes : émissions électromagnétiques aux harmoniques de la fréquence d'horloge, bruitage sur l'alimentation, temps de calcul au pire cas, consommation dynamique importante, faible plage de fonctionnement du circuit (tension d'alimentation, température). De plus, la conception de l'arbre d'horloge, qui achemine le signal d'horloge à chaque composant de la puce, est de plus en plus complexe à mesure que les circuits contiennent plus de transistors.

Il existe une autre solution au problème de la synchronisation au sein d'une puce, qui consiste à faire de la synchronisation locale, à l'aide d'un protocole poignée de mains : ce sont les circuits asynchrones. Sur le papier, les circuits asynchrones ont de nombreux avantages : calcul en temps moyen et non pire cas, faible consommation dynamique (puisque seules les parties du circuit nécessaires au calcul de la sortie subissent des transitions), modularité, plage de fonctionnement étendue.

Cependant, dans les faits, l'industrie reste quasi-exclusivement synchrone. Ceci peut s'expliquer par plusieurs facteurs.

Tout d'abord, il existe le principe de frein technologique : tous les étudiants, dans toutes les filières de microélectronique apprennent à concevoir des circuits synchrones. La discrétisation du temps est un principe de base, considéré comme naturel, et le remettre en cause une fois qu'on a pris l'habitude est très difficile : cela demande de revoir de nombreuses techniques.

Ainsi, pour une société, passer du développement de circuits synchrones au développement de circuits asynchrones demanderait d'acquérir des compétences supplémentaires, compétences très rares à trouver puisque les personnes formées à l'asynchrone sont rares.

De plus, si sur le papier les circuits asynchrones ont de nombreux avantages par rapport aux circuits synchrones, en pratique et en l'état actuel de la recherche, ils ont un inconvénient de taille : ils sont plus gros que leur équivalent synchrone. En effet, un circuit synchrone ne calcule qu'une fonction logique, alors que le circuit

asynchrone calcule, en plus, la complétion du calcul, ainsi que l'acquittement des entrées. Ceci est d'autant plus vrai pour les circuits quasi insensibles aux délais, qui ne font aucune hypothèse temporelle, et qui s'adaptent donc à la vitesse effective du circuit, selon les conditions de son environnement (température, tension d'alimentation).

Pour que l'industrie puisse migrer de manière non-anecdotique vers une technologie asynchrone, il est nécessaire d'avoir des outils permettant le développement de ces circuits, de manière automatisée, fiable, et qui permettent d'obtenir des circuits suffisamment optimisés pour être intéressants, par rapport à leurs équivalents synchrones. À l'heure actuelle, de tels outils manquent cruellement : il existe quelques outils de synthèse de circuits asynchrones, mais aucun ne propose une synthèse entièrement automatisée, à partir d'un langage de description permettant de modéliser tous les circuits.

Motivations

Le projet TAST, dans lequel s'intègre cette thèse, a pour but de fournir un ensemble d'outils permettant le développement de circuits asynchrones de manière équivalente à ce qui existe pour le synchrone. Dans le cadre de ce projet, cette thèse étudie l'automatisation du processus de synthèse et d'optimisation de circuits quasi insensibles aux délais.

Les circuits quasi insensibles aux délais sont les circuits les plus robustes : ils font le minimum d'hypothèses temporelles. Cette robustesse permet un développement modulaire des circuits : si l'on développe deux circuits quasi insensibles aux délais séparément, et que chacun implémente correctement le même protocole poignée de mains, alors on est assuré que le circuit obtenu en les réunissant fonctionnera correctement. Contrairement à un circuit synchrone, aucune étape de calcul des délais n'est nécessaire pour valider le bon fonctionnement du circuit.

Cependant, si l'on veut se passer de l'étape de calcul des délais dans le flot de synthèse, il faut s'assurer que le circuit est bien quasi insensible aux délais. Or, la vérification de cette propriété est très délicate. L'un des buts de la thèse est de prouver formellement que les circuits synthétisés par l'outil développé sont forcément quasi insensibles aux délais : ainsi, il est possible de s'affranchir du calcul des délais, sans avoir à effectuer de vérification de quasi insensibilité, puisque cette propriété est garantie formellement.

Contribution

Ce travail de thèse se base sur de précédents travaux de recherche ; cependant il apporte des contributions.

Tout d'abord, concernant le modèle de MDD décrit au chapitre 3, il existe dans la littérature une structure de diagrammes de décision multi-valués [1,2]. Mais cette structure n'a rien à voir avec la modélisation de bloc logique quasi insensible aux délais : elle ne modélise qu'une fonction logique à plusieurs variables multi-valuées. L'adaptation aux circuits quasi insensibles aux délais, et en particulier la spécification de la sémantique de consommation des entrées et de production des sorties

(problématique spécifique à l'asynchrone), est ma contribution personnelle.

De plus, j'ai légèrement modifié cette structure, en ajoutant les nœuds fourche, afin de permettre plus facilement la mise en commun de nœuds entre plusieurs sorties.

De même les algorithmes d'optimisation sur ce modèle sont ma contribution ; ils sont pour certains inspirés d'algorithmes existants, modifiés pour s'adapter aux particularités du modèle, dues aux contraintes de l'asynchrone quasi insensible aux délais.

Enfin, j'ai rélisé le calcul des probabilités de transition des nets sur le MDD, et ainsi l'équilibrage des arbres de portes OR en fonction de ces probabilités.

Concernant la synthèse de circuit, deux algorithmes sont proposés dans ce manuscrit. La synthèse DIMS, basique, existe depuis longtemps dans la littérature : c'est la façon la plus simple de générer un circuit quasi insensible aux délais.

La synthèse en portes deux entrées, présentée ensuite, est entièrement de mon fait.

L'une des composantes majeures de ce travail de thèse est la formalisation de la synthèse, et la preuve formelle que le flot de synthèse produit des circuits quasi insensibles aux délais. Ce travail théorique, présenté au chapitre 5, est entièrement le mien.

De même, j'ai entièrement réalisé l'implémentation du flot de synthèse décrit dans ce document, au sein du projet TAST, ainsi que l'évaluation de ce flot de synthèse à l'aide de circuits d'exemple. Cependant, la génération du MDD à partir du programme CHP, ainsi que la projection technologique, le dimentionnement, et tout ce qui suit dans le flot de synthèse, font l'objet de travaux séparés, et ne sont pas de mon ressort.

Plan du manuscrit

Ce manuscrit se compose de six chapitres.

Le premier chapitre présente le fonctionnement des circuits asynchrones, et plus particulièrement des circuits quasi insensibles aux délais. Les concepts de base d'un circuit asynchrone sont d'abord définis : dans un circuit asynchrone, la synchronisation se fait à l'aide de canaux de communication, et d'un protocole poignée de mains.

Puis les différentes classes de circuit asynchrone sont présentées, en commençant par celle qui fait le moins d'hypothèses temporelles, la classe des circuits insensibles aux délais, et en ajoutant progressivement des hypothèses, jusqu'aux circuits micropipeline et aux circuits de Huffman. Enfin, les circuits quasi insensibles aux délais sont étudiés en détail.

Le deuxième chapitre présente les différentes techniques de synthèse de circuits asynchrones qui existent. Il classe ces techniques en trois catégories.

D'un côté, les synthèses dirigées par la syntaxe, qui implémentent chaque construction du langage de spécification par un bloc du circuit : le circuit conserve la structure du programme d'entrée.

Ensuite, sont présentées les techniques à base de compilation : le programme spécifiant le circuit est compilé dans un ou plusieurs formats intermédiaires, transformé, optimisé, puis synthétisé sous forme de circuit. Ces techniques, plus complexes, donnent des circuits mieux optimisés, du fait de la granularité plus fine de l'optimisation.

Enfin, sont vues les techniques originales, qui ne rentrent pas dans les catégories ci-dessus.

Le troisième chapitre présente les différents modèles de circuits asynchrones utilisés dans cette thèse.

Tout d'abord, les modèles comportementaux. Le langage CHP, qui est le format d'entrée de TAST, modélise un circuit sous forme de processus communicants qui sont utilisés dans de nombreux outils de synthèse asynchrone. Tous les programmes CHP ne sont pas synthétisables : il y a des règles à respecter, notamment sur l'opérateur séquentiel, qui spécifie qu'une action doit être effectuée après une autre. Un programme qui respecte ces règles est dit DTL [3].

Le deuxième modèle étudié est le modèle de MDD (Diagramme de Décision Multivalué). Ce modèle a été élaboré au cours de cette thèse. Il décrit le comportement du circuit d'une manière adaptée à la vérification de la quasi insensibilité.

Ensuite, viennent les modèles structurels. La netlist est la structure classique permettant de décrire un réseau de portes logiques interconnectées, qui implémente un circuit. Une netlist peut être rendue hiérarchique, c'est-à-dire que certaines portes peuvent être elles-mêmes implémentées à l'aide d'une netlist. La hiérarchie est très utile lors du développement d'un circuit, afin de factoriser les fonctions communes de différentes parties du circuit : c'est l'équivalent du découpage fonctionnel d'un logiciel.

Le dernier modèle utilisé dans cette thèse est composé de règles de production. Ce modèle permet de décrire une netlist de manière formelle, en spécifiant le comportement de chaque porte de la netlist.

Le quatrième chapitre présente la technique de synthèse et d'optimisation de circuits QDI développée au cours de cette thèse. Pour cela, une première méthode de synthèse à partir de MDD, naïve, est présentée : c'est la synthèse DIMS. Puis, on présente les techniques d'optimisation du MDD.

Une méthode de synthèse plus efficace est ensuite présentée. Cette méthode permet de synthétiser des netlists dont le fan-in est borné (seulement des portes à deux entrées), et ainsi il est possible d'effectuer une projection technologique ne faisant que des fusions de portes, et respectant donc la quasi insensibilité aux délais du circuit.

Enfin, le calcul sur le MDD des probabilités de transition est présenté. Ce calcul permet d'équilibrer les arbres de portes OR lors de la synthèse.

Le Cinquième chapitre présente la partie théorique de cette thèse : la preuve formelle qu'un circuit synthétisé à partir d'un MDD à l'aide de la technique décrite dans le chapitre précédent est quasi insensible aux délais, sous réserve que le MDD satisfasse certaines contraintes. Cette preuve est basée sur le modèle de netlist.

Le Sixième et dernier chapitre présente l'implémentation de cette technique dans TAST, et l'évalue sur des circuits d'exemple. Les circuits d'exemple sont des opérateurs arithmétiques. Les MDDs sont générés automatiquement, à partir d'un pro-

gramme externe. Chaque circuit est synthétisé en différentes tailles, en faisant varier le nombre de bits, afin d'obtenir un ensemble de circuits de taille conséquente.

Chapitre 1

Les circuits asynchrones Quasi Insensibles aux Délais

1 Les circuits asynchrones

Un circuit intégré est constitué d'un ensemble de blocs logiques communiquant entre eux. Pour que deux blocs logiques puissent communiquer, il faut qu'ils soient synchronisés. Il y a deux manière de synchroniser les blocs logiques entre eux :

- soit de manière globale, à l'aide d'un signal l'horloge, qui indique à chaque bloc l'instant où il peut effectuer une communication. On parle alors d'un circuit synchrone,
- soit de manière locale, à l'aide d'un protocole de communication qui spécifie quand peut avoir lieu la communication. On parle alors d'un circuit asynchrone.

La quasi-totalité des circuits intégrés qui sont réalisés aujourd'hui sont synchrones. En effet, les outils de conception sont faits pour concevoir des circuits synchrones, et les industriels sont assez réticents à changer leurs outils et méthodes de conception. D'autant plus que les circuits synchrones sont en général plus petits que leurs équivalents asynchrones en nombre de portes.

De plus, la discrétisation du temps, hypothèse fondamentale que font les circuits synchrones, facilite la conception de ces circuits : qu'importe ce qu'il se passe entre deux fronts d'horloge, pourvu qu'au prochain front d'horloge la valeur calculée soit correcte et stabilisée. Les concepteurs de circuits asynchrones, eux, ne font pas d'hypothèse de discrétisation du temps. Ils doivent donc se soucier de tout ce qui se passe en sortie des blocs logiques, et en particulier garantir qu'il n'y aura pas d'aléa.

Cependant, cette hypothèse de temps discret, qui simplifie la conception des circuits synchrones, les ralentit considérablement : un bloc logique dans un circuit synchrone passe une bonne partie de son temps à attendre le prochain front d'horloge. En effet, quelles que soient les conditions (valeur des entrées, température, tension d'alimentation, ...), un bloc logique synchrone doit avoir eu le temps de produire sa sortie avant que le prochain front d'horloge ne se produise. Les blocs sont dimensionnés de manière à garantir cette hypothèse : ils sont dimensionnés pour l'hypothèse du pire cas, même si cette éventualité n'arrive que très rarement. Ainsi, la plupart du temps, le bloc logique a fini son calcul bien avant le front d'horloge, et ne fait plus rien jusqu'à ce que celui-ci arrive.

1.1 Intérêt des circuits sans horloge

Les circuits synchrones sont ultra-majoritairement utilisés dans le monde de la micro-électronique. Cependant, la présence du signal global d'horloge, qui est fondamental dans le modèle de circuits synchrones, pose un certain nombre de problèmes. En effet, ce signal doit arriver en même temps à tous les blocs logiques du circuit, ce qui nécessite de concevoir un arbre d'horloge parfaitement équilibré. Plus la fréquence des circuits augmente, plus l'équilibrage de cet arbre est critique. Un circuit sans horloge est conçu sans arbre d'horloge.

De plus, la présence d'une fréquence d'horloge impose que pour chaque bloc du circuit, le délai du chemin critique ne puisse dépasser le délai entre deux fronts d'horloge. Ainsi, un circuit synchrone travaille toujours au pire cas, quelles que soient les données à traiter. On peut cependant noter qu'il existe des techniques, telles le retiming, qui permettent d'équilibrer les délais des différents chemins dans le circuit, ce qui réduit l'impact de ce problème.

Au contraire, dans un circuit asynchrone, le temps de calcul d'un bloc combinatoire s'adapte aux données à traiter. Ainsi, le temps pertinent est le temps moyen, et non le délai du pire cas.

De plus, cette contrainte de délai du chemin critique impose une plage restreinte de température et de tension d'alimentation, puisque le délai du chemin critique dépend de ces deux grandeurs.

Cette limitation n'existe pas intrinsèquement pour les circuits asynchrones — en fait, elle dépend du modèle de circuit asynchrone utilisé, et des hypothèses temporelles que pose celui-ci. Les circuits quasi insensibles aux délais (cf. section 2.2), dont il est question dans cette thèse, ne sont pas sujets à cette limitation, et ont une plage de fonctionnement très étendue.

Pour finir, le fait de forcer tout le circuit à travailler à une fréquence précise génère du bruit à cette fréquence (et à ses harmoniques) sur la ligne d'alimentation, ainsi qu'une émission électromagnétique. Ceci est utilisé dans certaines attaques : connaissant la structure du circuit, on peut déceler l'activité d'une zone de la puce en fonction des données reçues.

En résumé, dans un circuit asynchrone, la synchronisation entre blocs est faite localement. Chaque bloc travaille à sa vitesse, qui dépend des données, le bruit causé sur la ligne d'alimentation et les émissions électromagnétiques sont donc répartis sur toute la plage de fréquences, et sont donc beaucoup plus difficiles à détecter.

1.2 Synchronisation locale et canaux de communication

Tout comme un circuit synchrone, un circuit asynchrone est constitué de blocs, qui calculent les valeurs de leurs sorties en fonction des valeurs de leurs entrée, et éventuellement de variables d'état internes.

Mais, pour gérer la notion de synchronisation locale entre ces blocs, on ajoute la notion de canaux de communication, qui connectent les blocs entre eux et leur permettent de communiquer. La figure 1.1 illustre la structure d'un circuit asynchrone au niveau des blocs et des canaux de communication.

Un canal de communication comporte 3 éléments :

– les données à transmettre,

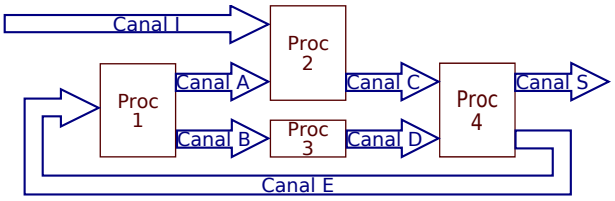


FIG. 1.1 – Illustration de la structure d'un circuit asynchrone, composé de blocs logiques communiquant via des canaux de communication.

- un signal de requête
- un signal d'acquittement

Le signal de requête indique au bloc récepteur que les données sont disponibles. Il est dans le sens des données, c'est-à-dire émis par le bloc émetteur et reçu par le bloc récepteur Le signal d'acquittement indique au bloc émetteur que les données ont bien été reçues. Il est dans le sens opposé aux données, c'est-à-dire émis par le bloc récepteur (du canal) et reçu par le bloc émetteur (du canal).

Ces deux signaux permettent de déterminer à quel instant la communication peut être effectuée, c'est-à-dire à quel instant le bloc émetteur peut modifier la valeur sur le canal, et à quel instant le bloc récepteur peut lire la valeur sur le canal. Tout ceci est spécifié par un protocole poignée de mains.

1.3 Protocole poignée de mains

Le protocole poignée de mains définit comment les signaux de requête et d'acquittement interagissent entre eux, à quel instant la donnée est accessible. Il est défini à partir de la notion d'événements sur les signaux de requête et d'acquittement. Un événement correspond à une transition sur le signal.

Le protocole est défini de la manière suivante :

- Un événement sur le signal de requête indique que les données sont disponible sur le canal, le bloc récepteur peut lire la valeur du canal et commencer son calcul.
- Lorsque le bloc récepteur n'a plus besoin de la donnée présente sur le canal, un événement sur le signal d'acquittement est émis.
- le protocole recommence alors, avec un nouvel événement sur le signal de requête.

Comme les signaux de requête et d'acquittement sont booléens, il y a deux types d'événement qui peuvent avoir lieu sur un signal : une transition montante (passant de la valeur logique 0 à la valeur logique 1), ou une transition descendante (passant de la valeur logique 0 à la valeur logique 1). Ces événements sont bien entendu entrelacés : une transition descendante suit forcément une transition montante.

Il y a donc deux choix possibles sur la manière dont les transitions de la requête et de l'acquittement s'enchaînent : soit les transitions du signal d'acquittement suivent les transitions du signal de requête dans le même sens, soit dans le sens opposé. Pour des raisons d'optimisations possibles du circuit obtenu, c'est le second choix qui est fait, comme présenté figure 1.2.

Il reste encore plusieurs façons d'implémenter le protocole poignée de mains, selon les événements que l'on considère comme faisant avancer le protocole poignées

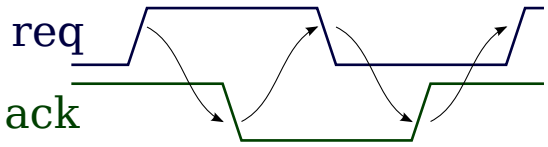


FIG. 1.2 – Le protocole poignée de mains

de mains.

Protocole deux phases

La manière la plus simple d'implémenter le protocole est de considérer toutes les transitions comme des événements du protocole poignée de mains.

On obtient ainsi deux phases :

1. **requête** : une donnée est disponible sur le canal
2. **acquittement** : la donnée a été traitée et peut être modifiée

Ces phases correspondent aux deux phases du protocole poignée de mains. Le protocole deux phases est présenté figure 1.3.

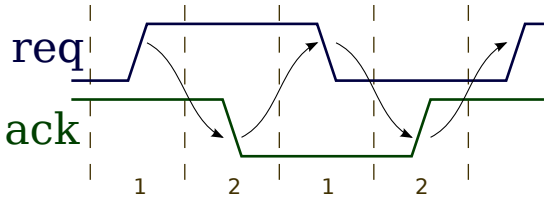


FIG. 1.3 – Le protocole deux phases

Cependant, considérer indifféremment les transitions montantes et descendantes induit une asymétrie entre deux itérations du protocole poignée de mains, qui se traduit par une complexité au niveau du circuit dont on souhaiterait se passer. De ce fait, le protocole deux phases est peu utilisé.

Protocole quatre phases

Pour implémenter le protocole poignée de mains, on peut aussi ne considérer qu'un seul type de transition pour la requête et pour l'acquittement : uniquement les transitions montantes, ou uniquement les transitions descendantes. Ainsi, les itérations successives du protocole poignée de mains sont toutes symétriques, et le circuit réalisé est moins complexe.

On obtient ainsi quatre phases : deux phases de calcul pendant lesquelles se déroule le protocole poignées de main, et deux phases de réinitialisation (ou remise à zéro, RAZ), destinées à revenir dans un état où le protocole poignées de main pourra continuer, comme le présente la figure 1.4 :

- **requête** : une donnée est disponible sur le canal
- **acquittement** : la donnée a été traitée et peut être modifiée
- **réinitialisation de la requête** : la donnée n'est plus nécessaire
- **réinitialisation de l'acquittement** : le canal est prêt pour une nouvelle communication

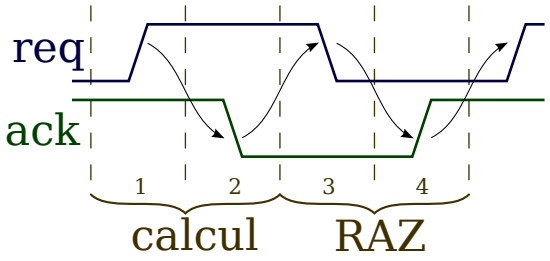


FIG. 1.4 – Le protocole quatre phases

Du fait de la symétrie entre les itérations successives du protocole poignée de mains, et de la simplicité des circuits qui l'implémentent, le protocole quatre phases est le plus utilisé. Dans la suite de cette thèse, c'est toujours le protocole quatre phases qui est retenu.

2 Classes de circuits asynchrones

Les canaux de communication, munis d'un protocole poignée de mains, permettent d'implémenter des circuits asynchrones, en fournissant un mécanisme de synchronisation local entre les blocs logiques.

Dans un circuit synchrone, pour que la synchronisation entre les blocs logiques fonctionne, des hypothèses sur les délais de calculs dans les blocs logiques sont faites : le délai entre deux registres doit toujours être inférieur à la période de l'horloge, sinon le circuit ne fonctionne pas.

Les circuits asynchrones n'ont pas besoin de faire d'hypothèse temporelles aussi forte. On distingue plusieurs modèles de circuits asynchrones, selon les hypothèses temporelles faites pour garantir que le circuit fonctionne.

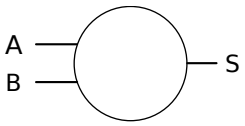
2.1 Circuits Insensibles aux Délais

Les circuits qui font le moins d'hypothèses temporelles sont les circuits Insensibles aux Délais [4]. En fait, aucune hypothèse n'est faite. Par définition, un circuit insensible aux délais aura le même comportement quels que soient les délais de ses portes et de ses fils. Pour cela, la notion de causalité entre les événements (transitions) est importante : chaque transition en entrée d'un élément doit causer une transition en sortie. On dit que la transition en sortie acquitte la transition en entrée.

Cependant, les limitations des circuits DI sont très handicapantes. En effet, dans un circuit DI, une porte ne peut pas changer sa sortie avant que toutes ses entrées aient elles-mêmes changé. Dès lors, la seule porte à une sortie et plusieurs entrées qui puisse être utilisée est la porte de Muller, présentée figure 1.5 [5]. Le problème vient du fait que les fonctions réalisables uniquement avec des portes de Muller (et des inverseurs...) sont très limitées [5]. La seule solution est alors d'avoir recours à des portes complexes.

A	B	S
0	0	0
0	1	S^{-1}
1	0	S^{-1}
1	1	1

(a) table de vérité



(b) représentation

FIG. 1.5 – La porte de Muller

2.2 Circuits Quasi Insensibles aux Délais

On a vu ci-dessus que la classe des circuits Insensibles aux Délais était trop restreinte pour être utilisable dans la plupart des cas. Les circuits Quasi Insensibles aux Délais (QDI)font la plus petite hypothèse temporelle nécessaire pour que l'on puisse implémenter tous les circuits possibles : on fait l'hypothèse que certaines fourche du circuit sont isochrone.

Une fourche est un fil connecté à l'entrée de plusieurs portes, comme illustré figure 1.6. Elle est isochrone si l'on fait l'hypothèse que les délais de ses branches sont égaux, c'est-à-dire qu'une transition sur la fourche arrivera en même temps à l'entrée de chacune des portes auxquelles elle est connectée [6, 7].

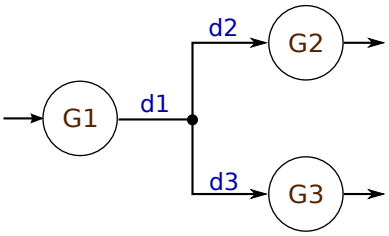


FIG. 1.6 – La fourche est isochrone si les délais $d2$ et $d3$ sont égaux.

L'hypothèse de fourches isochrones a des conséquences importantes sur le modèle. En effet, elle permet de n'acquitter qu'une des branches de ces fourches, puisque l'on suppose que la transition s'est propagée de la même façon dans toutes les branches. Ainsi, on peut construire tous les circuits que l'on veut, comme montré dans [8]. De plus, L'hypothèse de fourche isochrone est la plus faible à apporter aux circuits DI pour que l'on puisse n'utiliser que des portes à une sortie ([9]).

En pratique, l'hypothèse de fourche isochrone n'est pas très contraignante. En effet, il faudrait que la différence de délais entre les branches soit très différente pour que le circuit ne fonctionne plus : de l'ordre du délai de propagation d'une transition dans toute la boucle qui implémente le protocole quatre phases, à travers l'acquittement. Une conception soignée permet donc de respecter cette hypothèse, par exemple en contraignant l'outil de routage.

De plus, les circuits quasi insensibles aux délais ne nécessitent pas d'analyse de timing pour vérifier qu'ils fonctionnent correctement. Ceci induit un gain de temps de conception considérable, en particulier grâce à la modularité que cela procure : un bloc fonctionnel peut être réutilisé sans se poser de questions, il fonctionnera quel que soit son environnement, du moment qu'il respecte le protocole poignée de

mains (et qu'il respecte les hypothèses éventuelles que requiert l'implémentation du bloc).

Ceci rend aussi les circuits quasi insensibles aux délais plus faciles à synthétiser automatiquement, puisqu'une étape difficile à automatiser (la vérification des timings) est supprimée.

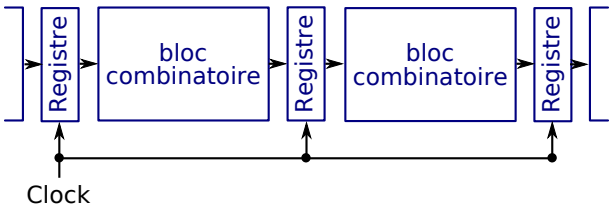
2.3 Circuits Indépendants de la Vitesse

Les circuits Indépendant de la vitesse (SI) font l'hypothèse que les délais dans les fils sont négligeables. Cette hypothèse revient à supposer que toutes les fourches du circuit sont isochrones. Ainsi, les circuits SI font légèrement plus d'hypothèses temporelles que les circuits QDI.

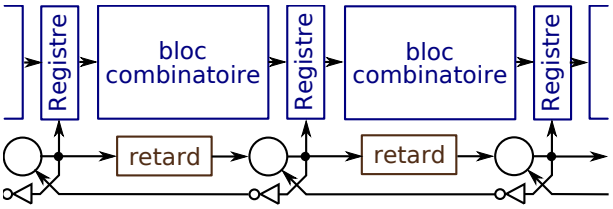
À part le fait que les circuits SI n'identifient pas les fourches isochrones des fourches sur lesquelles aucune hypothèse n'est nécessaire, les deux modèles sont équivalents. De fait, les circuits QDI sont plus pertinents, puisque davantage d'information est disponible. Ils sont donc plus utilisés.

2.4 Circuits Micropipeline

Les circuits micropipeline sont les circuits asynchrones plus proches des circuits synchrones. Tout comme les circuits synchrones, ils sont composés d'un ensemble de blocs combinatoires interconnectés par des registres, comme le montre la figure 1.7(a). La différence se situe au niveau de l'arbre d'horloge, qui est remplacé par un circuit calculant les signaux de requête et d'acquittement, présenté figure 1.7(b).



(a) circuit synchrone



(b) circuit micropipeline

FIG. 1.7 – Un circuit micropipeline a la même forme qu'un circuit synchrone : il se compose d'un ensemble de blocs combinatoires, interconnectés par des registres. Dans le circuit synchrone, à chaque front (front montant) d'horloge, une donnée avance d'un registre ; alors que dans le circuit micropipeline, l'arbre d'horloge est remplacé par des signaux de requête et acquittement, qui implémentent le protocole poignée de mains.

Les circuits micropipeline combinent les avantages des circuits synchrones pour le chemin de données (nombreux outils disponibles pour les concevoir, les optimiser, les

caractériser) avec certains avantages des circuits asynchrones (moindres contraintes dues à l'arbre d'horloge, moins de bruits parasites à une fréquence fixe). Cependant, du fait des blocs retard qui sont fixes, il fonctionnent aussi au pire cas puisqu'il ne faut pas que le délai du bloc combinatoire soit supérieur au bloc retard qui lui correspond. Ces circuits nécessitent donc une analyse des délais pour vérifier leur bon fonctionnement.

2.5 Circuits de Huffman

Ces circuits sont spécifiés sous la forme d'un automate (ou machine à états) fini asynchrone. Ce sont les plus proches des circuits synchrones. Il n'y a pas d'horloge globale qui cadence l'évolution du circuit, mais on introduit une borne sur les délais du calcul de l'état suivant, ainsi que sur les entrées et sorties du circuit.

La synthèse de ces circuits se décompose en trois étapes, comme pour les circuits synchrones :

- minimisation de l'espace d'états
- codage binaire des états
- optimisation logique

La figure 1.8 représente la structure d'un circuit de Huffman.

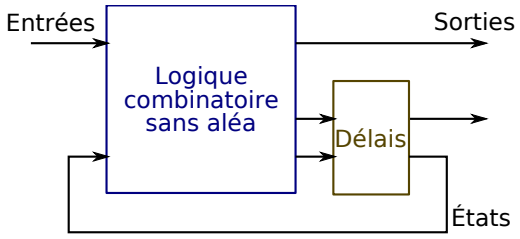


FIG. 1.8 – Structure d'un circuit de Huffman.

3 Quasi insensibilité aux délais

3.1 Intérêt de l'insensibilité aux délais

Les circuits synthétisés dans cette thèse sont tous quasi insensibles aux délais. La quasi insensibilité aux délais est la classe de circuits la plus robuste, excepté les circuits insensibles aux délais.

En effet, les circuits quasi insensibles aux délais fonctionnent quels que soient les délais dans les portes et dans les fils du circuit. Ainsi, la température et la tension d'alimentation, qui ont une incidence directe sur ces délais, n'empêchent pas le circuit de fonctionner (dans les limites de la technologie, bien entendu, des conditions extrêmes pouvant détruire le matériel).

De même, le circuit fonctionne malgré la variation technologique (c'est-à-dire le fait que les éléments du circuit gravé n'ont pas exactement les caractéristiques théoriques, ce qui a une incidence sur les délais effectifs qui sont différents des délais simulés)

Pour finir, le circuit fonctionne quels que soient les délais. Il n'est donc pas nécessaire de faire l'analyse des délais pour valider le fonctionnement du circuit. Du

moment que les blocs respectent le protocole poignée de mains, le circuit fonctionnera correctement. Il est ainsi facile de concevoir des circuits modulaires, composés de blocs distincts, testés séparément. L'assemblage de ces circuits fonctionnera, sans que l'on n'ait besoin de valider le comportement global, comme ça serait le cas avec des circuits faisant plus d'hypothèses temporelles.

3.2 Fourche isochrone et Turing-complétude

La seule hypothèse temporelle que font les circuits quasi insensibles aux délais est que certaines fourches sont isochrones, c'est-à-dire que leurs branches ont le même délai (comme expliqué section 2.2).

Cette hypothèse suffit à avoir une classe de circuits Turing-complète : on peut implémenter une machine de Turing avec un circuit quasi insensibles aux délais [8]. Concrètement, cela signifie que l'on peut implémenter tous les circuits que l'on veut.

3.3 Aléas

Un aléa est une activité non désirée en réponse à un changement sur certaines entrées [10]. La figure 1.9 représente un exemple d'aléa.

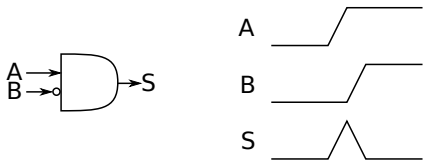


FIG. 1.9 – Exemple d'aléa. Une transition montante sur A et sur B crée un aléa : la sortie S passe temporairement à 1 avant de redescendre à 0.

Dans un circuit synchrone, les aléas sont inoffensifs : ce qui compte, c'est la valeur de la sortie au moment du front de l'horloge. Un aléa se produit pendant le calcul. L'horloge est calculée pour laisser au circuit le temps de se stabiliser. Par conséquent, les aléas ne sont pas échantillonnés.

Cependant, dans un circuit asynchrone, une transition sur la requête ou l'acquittement est interprétée comme une phase du protocole poignée de mains, et a donc une incidence directe sur le fonctionnement du circuit. Ainsi, un aléa dans un circuit risque fortement de bloquer le bon fonctionnement du circuit. Un circuit asynchrone doit donc garantir qu'il ne génère pas d'aléa.

La porte de Muller (présentée figure 1.5(b)), qui est utilisée pour construire les circuits QDI, ne produit pas d'aléa. En effet, cette porte effectue un rendez-vous : pour qu'une transition ait lieu à sa sortie, il faut qu'il y ait eu une transition sur chacune de ses entrées. Ainsi, durant une phase de calcul, il ne peut pas y avoir plus d'une transition sur chacune de ses entrée, et il ne peut donc pas y avoir plus d'une transition sur sa sortie. La porte ne produit donc pas d'aléa.

3.4 Codage des données insensible aux délais

Dans le modèle du canal de communication présenté section 1.2, une hypothèse temporelle est faite. En effet, le signal de requête indique que des données sont

disponibles sur le canal. Mais ce n'est pas parce que le canal émetteur a émis les données avant le signal de requête que le bloc récepteur reçoit les données avant la requête. Dans un circuit quasi insensible aux délais, il n'y a aucune garantie sur les délais de propagations dans deux fils distincts.

Pour garantir que cette hypothèse est bien respectée sans faire d'hypothèses sur les délais de propagations dans les fils, on utilise un codage des données insensible aux délais [11, 12]. Ce codage code en même temps le signal de requête et la donnée. Ainsi, il est garanti que les deux informations arrivent forcément en même temps. La figure 1.10 présente un codage insensible aux délais : le codage 1 parmi n, ou multi-rails.

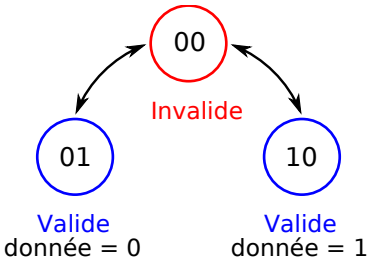


FIG. 1.10 – Exemple de codage insensible aux délais : 1 parmi 2. La combinaison où les deux rails sont à 1 est interdite.

Le codage 1 parmi n est défini comme suit :
- Pour coder n valeurs, on utilise n fils, appelés «rails».
- Chaque rail correspond à l'une des valeurs possible.
- Lorsque le signal de requête est à 0, tous les rails sont à 0. On dit que le canal est invalide. Aucune donnée n'est disponible.
- Lorsque le signal de requête est à 1, l'un des rails est à 1, et les autres sont à 0. Le rail à 1 est celui correspondant à la donnée transmise sur le canal. On dit que le canal est valide.
- Les autres combinaisons, où plusieurs rails sont à 1 en même temps, sont interdites.

Ce codage est insensible aux délais. En effet, une seule transition indique à la fois un événement sur la requête et la valeur de la donnée transmise. Il y a donc forcément simultanéité des deux informations.

3.5 Mode *push/pull*

Toutes les considérations sur le protocole qui précèdent ne sont valables qu'en régime permanent : c'est-à-dire que l'on fait l'hypothèse que le circuit vient de terminer une itération du protocole avant d'en commencer une nouvelle. Cette hypothèse est bien entendu fausse lors de la première itération du protocole.

En fait, il faut choisir la phase du protocole quatre phases par laquelle on va commencer. A priori, il y a quatre choix possibles, un par phase. Mais initier le protocole à la phase 2 ou 3 correspondrait à avoir déjà émis la requête, ce qui n'est en général pas intéressant, puisque l'on ne sait pas si le bloc va tout de suite émettre une requête sur tous ses canaux de sortie Il reste alors deux choix :

- Soit on initie le protocole sur la phase 1. La première transition se fait alors sur le signal de requête, on dit que le canal est en mode *push* (ou encore que le port de sortie est actif, et que le bloc d'entrée est passif).
- Soit on initie le protocole sur la phase 4. La première transition se fait alors sur le signal d'acquittement, on dit que le canal est en mode *pull* (ou encore que le port d'entrée est actif, et que le port de sortie est passif).

Ces deux choix ont un intérêt, aussi sont-ils tous les deux utilisés. En effet, certains circuits testent la disponibilité de leurs entrées, sans attendre que l'entrée soit disponible si elle ne l'est pas : le bloc connecté en entrée est-il prêt à émettre une donnée ? On parle alors de sonde sur les entrées. Les canaux d'entrée d'un tel circuit sont forcément en mode *push*, puisque ce circuit veut savoir si une donnée est disponible sans avoir à la demander.

De la même manière, un circuit peut tester la disponibilité sur ses sorties : le bloc connecté en sortie est-il prêt à recevoir une donnée ? Les canaux de sortie d'un tel circuit sont forcément en mode *pull*, puisque ce circuit veut savoir si une donnée peut être émise sans émettre de requête.

3.6 Reset

Dans tout ce qui précède, une hypothèse a été faite implicitement : les signaux de requête et d'acquittement de tous les canaux ont initialement la bonne valeur, avant que le protocole commence. Le tableau 1.1 résume les valeurs initiales des signaux de requête et d'acquittement, selon le mode *push*/*pull* du canal.

Pour que cette hypothèse soit vérifiée, il faut un mécanisme qui positionne ces signaux à la bonne valeur. Un signal global nommé `Reset` est utilisé pour ceci. Lorsque ce signal est actif, le protocole est bloqué, et les signaux prennent leur valeur initiale, suivant le tableau 1.1. Lorsqu'il est inactif, le protocole se déroule normalement.

	push	*pull*
req.	$0\nearrow$	0
ack.	1	$0\nearrow$

TAB. 1.1 – Valeurs des signaux de requête et d'acquittement des canaux lors de la phase reset. «$\nearrow$» représente la première transition après reset, qui initie le protocole quatre phases.

3.7 Initialisation

Certains circuits, nécessitent qu'un bloc commence par émettre une requête sur un canal de sortie avant de passer à un comportement «permanent» (il émet les requêtes en sortie en fonction des requêtes en entrée). Ce comportement est nécessaire lorsqu'il y a une boucle dans le circuit, afin d'initier les requêtes dans la boucle. La figure 1.11 présente un exemple de circuit dans lequel un tel comportement est nécessaire : une machine à états, qui émet une séquence de valeurs sur le canal S.

Dans un tel cas, il est intéressant de pouvoir initier le protocole sur la phase 2 : on sait que l'on va commencer par émettre une requête, et on commence donc en

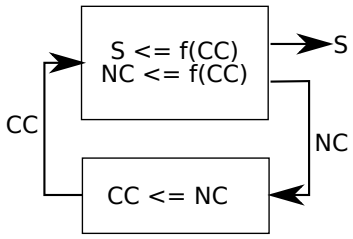


FIG. 1.11 – Exemple de circuit : une machine à états. Il faut que le canal CC (ou le canal NC) soit initialisé pour que le circuit fonctionne.

attendant que la requête soit acquittée. On dit dans ce cas que le canal est initialisé. On note que la valeur des signaux de requête et d'acquittement à Reset ne sont pas modifiées par l'initialisation. Ce qui change, c'est que la requête est émise dès que le signal `Reset` est relâché, au lieu d'attendre les données en entrée du bloc.

Pour des raisons de simplicité, et parce que cela ne ferait peu de sens autrement, on n'autorise que les canaux en mode *push* à être initialisés. Un canal en mode *pull* doit attendre son acquittement pour pouvoir émettre la requête, il n'y aurait pas d'intérêt à ce qu'il soit initialisé.

3.8 Vérification de la quasi insensibilité aux délais

Vérifier si un circuit donné est quasi insensibles aux délais est un problème coûteux [13] et difficile dans le cas général. En fait, c'est l'un des principaux problèmes qui motive ce travail de thèse.

Dans [4], Udding se base sur la théorie des traces, c'est-à-dire sur l'ensemble des séquences d'événements possibles, pour définir un circuit insensible aux délais. Mais cette méthode n'est pas implémentable : le nombre de séquences d'événements possibles dans un circuit explose avec la taille du circuit... Dans [8], une condition nécessaire et suffisante sur les règles de productions qui décrivent un circuit est donnée pour que ce circuit soit QDI. Les règles de production sont une manière de modéliser algébriquement un circuit : chaque fil du circuit est modélisé par un couple de règles de production, c'est-à-dire d'équations booléennes qui spécifient les conditions pour qu'il se produise une transition montante (respectivement descendante) sur le fil. Cependant, cette condition n'est pas directement vérifiable algorithmiquement.

Dans [13], une méthode est implémentée pour vérifier la quasi insensibilité aux délais de circuits combinatoires. Cette méthode est très coûteuse en temps de calcul : les chiffres donnés dans l'article correspondent à une dizaine de minutes pour un circuit de l'ordre du millier de portes, et dizaine d'heures de calcul pour des circuits atteignant la dizaine de milliers portes, la méthode n'étant bien entendu pas linéaire.

Ce travail de thèse adopte une nouvelle approche. J'ai développé un modèle, qui permet de spécifier un circuit. Ce modèle peut être vu comme un langage bas niveau de description comportementale de circuit. Un ensemble de contraintes sur ce modèle garantissent que le circuit décrit est QDI. Un algorithme permet de dériver directement le circuit à partir du modèle (c'est-à-dire de synthétiser le circuit, pour en obtenir une description structurelle).

Cette méthodologie permet de s'affranchir de l'étape de vérification de la quasi-insensibilité aux délais sur le circuit : cette vérification est faite sur le modèle, qui a

une structure adaptée à cette vérification. L'étape de synthèse qui génère le circuit à partir du modèle garantit que le circuit généré est bien QDI.

Ainsi, toutes les transformations à faire sur le circuit pour l'optimiser peuvent être faites sur le modèle, tout en préservant la propriété de quasi insensibilité aux délais.

3.9 Protocoles

Le protocole quatre phases spécifie le comportement des signaux de requête et d'acquittement de chaque canal d'entrée et de sortie d'un bloc.

Un bloc QDI peut comporter plusieurs entrées et plusieurs sorties ; il est donc connecté à plusieurs canaux. Il faut déterminer la synchronisation entre les phases du protocole pour les différents canaux. Pour cela, on définit plusieurs protocoles, qui spécifient la coordination entre les phases des entrées et les phases des sorties du protocole quatre phases.

Protocole séquentiel

Le protocole séquentiel est le protocole le plus simple : les phases du protocole quatre phases sont transmises de l'entrée à la sortie, de manière séquentielle. Ainsi, la communication entre l'entrée et la sortie est transparente. La figure 1.12 illustre les phases du protocole séquentiel :

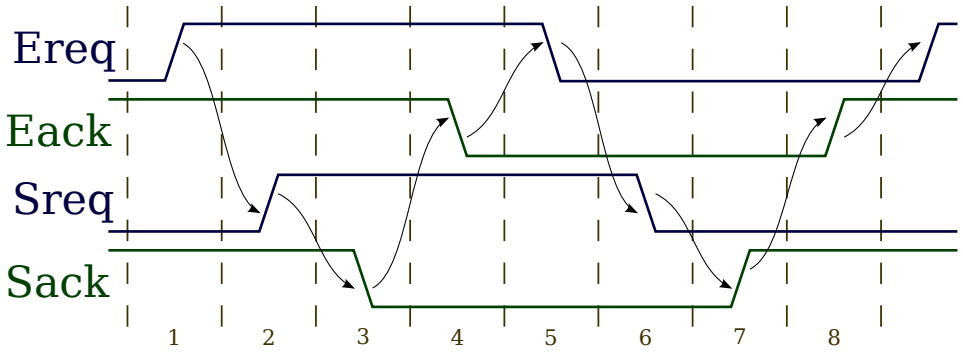


FIG. 1.12 – Dans le protocole séquentiel, les phases du protocole quatre phases sont transmises de l'entrée E à la sortie S de manière séquentielle

- **phase 1 de l'entrée** : activation du signal de requête et arrivée de la donnée,
- **phase 1 de la sortie** : la requête est propagée à la sortie et la donnée en sortie est calculée en fonction des données en entrée, et émise,
- **phase 2 de la sortie** : activation du signal d'acquittement,
- **phase 2 de l'entrée** : l'acquittement est propagé à l'entrée,
- **phase 3 de l'entrée** : remise à zéro de la requête,
- **phase 3 de la sortie** : la requête est remise à zéro,
- **phase 4 de la sortie** : remise à un de l'acquittement,
- **phase 4 de l'entrée** : l'acquittement est remis à un.

Le principal intérêt de ce protocole est sa grande simplicité, qui se traduit par un faible surcoût sur le circuit, en nombre de portes. Cependant, la séquentialité induit des temps d'attente qui ne sont pas nécessaires. Un circuit utilisant le protocole séquentiel est donc plus lent que s'il utilisait un protocole plus complexe.

De plus, si l'on connecte deux blocs séquentiels, les temps de cycles s'ajoutent, comme l'illustre la figure 1.13. En effet, le temps de cycle total d'un bloc séquentiel contient l'attente entre l'instant où la requête a été émise sur la sortie et l'instant où l'acquittement de la sortie est reçu. Ce temps d'attente correspond au temps de cycle du bloc récepteur.

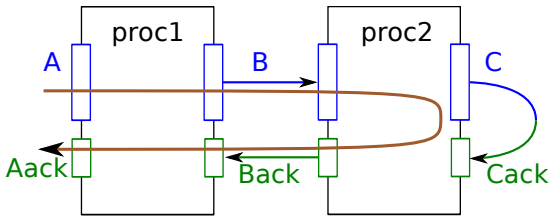


FIG. 1.13 – Le temps de cycle de deux blocs séquentiels connectés se cumule : le délai entre la requête sur A et l'acquittement de Aack contient le délai entre la requête sur B et l'acquittement de Back.

Protocole WCHB : Weak Condition Half Buffer

Le protocole WCHB parallélise les étapes du protocole quatre phases à l'entrée et à la sortie, afin de gagner en temps de cycle. Les phases de l'entrée et de la sortie sont décalées, de sorte qu'elles puissent se dérouler en même temps sans problème, comme l'illustre la figure 1.14.

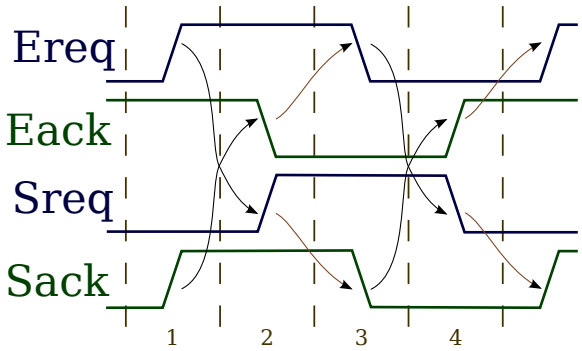


FIG. 1.14 – Le protocole WCHB. Les phases du protocole quatre phases à l'entrée et à la sortie se déroulent en parallèle, de manière décalée.

Les phases du protocole WCHB sont les suivantes :
- **phase 1 de l'entrée, phase 4 de la sortie** : réception d'une requête et arrivée de la donnée en entrée ; remise à un de l'acquittement en sortie (la sortie est donc disponible pour une nouvelle requête),
- **phase 2 de l'entrée, phase 1 de la sortie** : l'entrée est acquittée, la requête est propagée en sortie, et la donnée en sortie est calculée en fonction des données en entrée, et émise,
- **phase 3 de l'entrée, phase 2 de la sortie** : remise à zéro de la requête en entrée, et (indépendemment) activation du signal d'acquittement de la sortie,
- **phase 4 de l'entrée, phase 3 de la sortie** : remise à un de l'acquittement de l'entrée, et remise à zéro de la requête de sortie.

On peut noter que lors de la deuxième phase du protocole, l'entrée est acquittée en même temps que la valeur de sa donnée est utilisée pour le calcul. Ceci n'est possible que si le circuit contient une mémoire. En effet, si le circuit est purement combinatoire, il n'est pas capable de maintenir la valeur de la sortie (qui dépend de la valeur de l'entrée) alors que cette entrée a été acquittée, et est donc susceptible d'être remise à zéro. Une manière d'implémenter cette mémoire consiste à utiliser des portes de Muller, combinant la valeur à mémoriser avec le signal d'acquittement de la sortie. Ainsi, la valeur sera mémorisée par les portes de Muller tant que le signal d'acquittement de la sortie est à un, c'est-à-dire tant que la donnée en sortie doit être maintenue. On parle dans ce cas de demi-buffer (Half Buffer), puisque la valeur est maintenue pendant la moitié du cycle du protocole quatre phases.

Le protocole WCHB produit des circuits plus rapides que le protocole séquentiel, étant donné qu'il parallélise les phases entre l'entrée et la sortie, au prix d'un léger surcoût en nombre de portes, du à la mémorisation nécessaire. De plus, les temps de cycles ne se cumulent pas, comme le montre la figure 1.15. En effet, on peut voir un bloc WCHB comme deux demi-boucles qui s'intersectent. L'un correspond au chemin allant de la requête et la donnée d'entrée à l'acquittement de l'entrée, et l'autre au chemin allant de l'acquittement de la sortie à la requête et à la donnée de la sortie. Ces deux demi-boucles sont synchronisées, puisque la donnée émise à la sortie dépend de la donnée reçue en entrée. En connectant deux blocs WCHB, on forme une boucle complète. C'est le délai de cette boucle qui correspond au temps de cycle du circuit. Si l'on connecte plus de deux blocs, on obtient un ensemble de boucles. Le temps de cycle est alors le délai de boucle maximal.

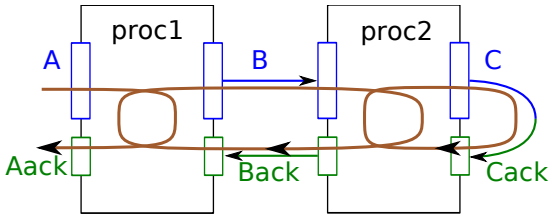


FIG. 1.15 – Le temps de cycle de deux blocs WCHB connectés ne se cumule pas. Chaque bloc est constitué de deux demi-boucles, qui combinées avec les demi-boucles des blocs connectés forment des boucles complètes. Le temps de cycle global est le maximum des temps de chaque cycle.

Protocole PCHB : PreCharge Half Buffer

Le protocole PCHB va plus loin dans la parallélisation. En effet, on peut remarquer que dans la phase de remise à zéro du protocole WCHB, la synchronisation entre l'entrée et la sortie est inutile : l'un peut se faire indépendemment de l'autre. Le protocole PCHB ne lève qu'à moitié cette synchronisation, en privilégiant la sortie, comme le montre la figure 1.16 : il permet à la requête de sortie d'être remise à zéro le plus tôt possible, indépendemment de l'état de l'entrée. La remise à un de l'acquittement de l'entrée, elle, est séquentielle.

Les phases du protocole PCHB sont les suivantes :

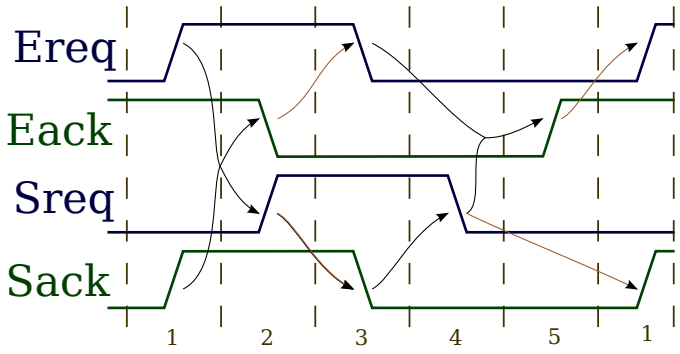


FIG. 1.16 – Le protocole PCHB. La remise à zéro de la requête de sortie est faite indépendemment de la remise à un de l'acquittement de l'entrée.

- **phase 1 de l'entrée, phase 4 de la sortie** : réception d'une requête et arrivée de la donnée en entrée ; remise à un de l'acquittement en sortie (la sortie est donc disponible pour une nouvelle requête),
- **phase 2 de l'entrée, phase 1 de la sortie** : l'entrée est acquittée, la requête est propagée en sortie, et la donnée en sortie est calculée en fonction des données en entrée, et émise,
- **phase 3 de l'entrée, phase 2 de la sortie** : remise à zéro de la requête en entrée, et (indépendemment) activation du signal d'acquittement de la sortie,
- **phase 3 de la sortie** : remise à zéro de la requête de sortie, indépendemment de l'entrée,
- **phase 4 de l'entrée** : remise à un de l'acquittement de l'entrée.

Ce protocole est plus rapide que le protocole WCHB, en particulier lorsque le bloc de sortie est rapide. En effet, la remise à zéro de la sortie est privilégiée. En contrepartie, ces circuits sont plus complexes à synthétiser. Ils nécessitent l'utilisation de portes de Muller asymétriques, et davantage de fils : le circuit synthétisé est plus gros.

Protocole PCFB : PreCharge Full Buffer

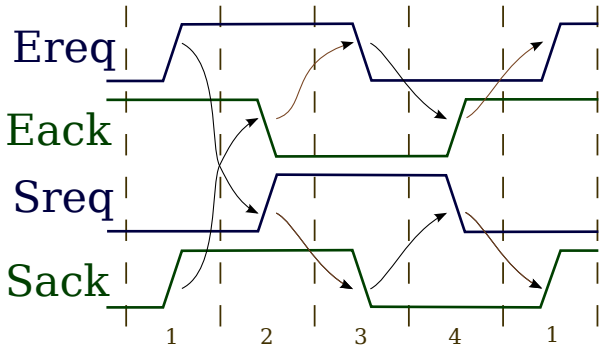


FIG. 1.17 – Le protocole PCFB

Les phases du protocole PCHB sont les suivantes :

- **phase 1 de l'entrée, phase 4 de la sortie** : réception d'une requête et arrivée de la donnée en entrée ; remise à un de l'acquittement en sortie (la sortie est donc disponible pour une nouvelle requête),

- **phase 2 de l'entrée, phase 1 de la sortie** : l'entrée est acquittée, la requête est propagée en sortie, et la donnée en sortie est calculée en fonction des données en entrée, et émise,
- **phase 3 de l'entrée, phase 2 de la sortie** : remise à zéro de la requête en entrée, et (indépendemment) activation du signal d'acquittement de la sortie,
- **phase 4 de l'entrée, phase 3 de la sortie** : remise à un de l'acquittement de l'entrée, indépendemment de la sortie, et remise à zéro de la requête de sortie, indépendemment de l'entrée.

Les circuits utilisant le protocole PCFB sont encore plus rapides que les circuits PCHB, et sont intéressants lorsque le bloc d'entrée est rapide : il n'est pas pénalisé par rapport à la sortie, contrairement au protocole PCHB. Cependant, ces circuits sont encore plus complexes.

Une étude complète de ces protocoles et de leur impact sur les performances d'un circuit asynchrone est en cours. L'analyse des performances d'un circuit asynchrone en fonction du protocole poignée de mains est un problème très complexe, mais d'un enjeu important [14].

4 Conclusion

Dans ce chapitre, on a présenté le principe de base des circuits asynchrones : un ensemble de blocs logiques synchronisés localement à l'aide de canaux de communications munis d'un protocole poignée de mains.

Cette synchronisation locale, qui définit les circuits asynchrones, peut être réalisée de plusieurs manières, faisant plus ou moins d'hypothèses temporelles sur le circuit : il existe plusieurs classes de circuits asynchrones. Parmi ces classes, les circuits quasi insensibles aux délais, qui sont les circuits utilisables faisant le minimum d'hypothèses temporelles, sont ceux que l'on cible dans cette thèse.

Dans les circuits quasi insensibles aux délais, la seule hypothèse temporelle qui est fait est celle des fourches isochrones : certaines fourches sont supposées isochrones, c'est-à-dire que les délais de propagation des transitions le long de leurs branches sont supposés être égaux. Afin de fonctionner sans hypothèse supplémentaire, les circuits quasi insensibles aux délais utilisent un codage des données insensible aux délais. Les codages retenus ici sont les codages 1-parmi-n.

Chapitre 2

Techniques de synthèse de circuits asynchrones

1 Introduction

Les circuits asynchrones présentent de nombreux atouts par rapport aux circuits synchrones, tels que disparition de l'arbre d'horloge, faible consommation dynamique, modularité, calcul en temps moyen, robustesse, etc. Pourtant, la quasi-totalité du monde industriel ne développe que des circuits synchrones. L'une des raisons est le manque d'outils de conception et de synthèse automatique pour les circuits asynchrones. La disponibilité de tels outils est un point crucial dans l'adoption par la communauté industrielle des technologies asynchrones.

Il n'est pas possible d'utiliser les techniques de synthèse «traditionnelles», c'est-à-dire conçues pour les circuits synchrones, pour synthétiser des circuits asynchrones. En effet, les techniques de synthèse synchrones ne se soucient pas des aléas, puisque le temps est discrétisé. Les circuits asynchrones les plus robustes ne tolèrent et ne produisent pas d'aléa. Leur fonctionnement requiert l'absence d'aléa pour fonctionner correctement. La technique de synthèse asynchrone que nous développons dans cette thèse nécessite de garantir que le circuit synthétisé est sans aléa. Ceci ne peut se faire qu'en mettant en place de nouvelles techniques de synthèse, basées dès la conception sur la génération de circuits sans aléa.

Ce chapitre présente les différentes techniques de synthèse de circuits asynchrones qui existent.

2 Synthèse dirigée par la syntaxe

La manière la plus simple de générer un circuit à partir d'un description comportementale est de diriger la synthèse par la syntaxe. Cette approche consiste à générer un circuit qui a une structure similaire au programme qui le décrit comportementalement. Ainsi, pour chaque construction du langage comportemental (opérateur, mot-clef, ...), un morceau de circuit est généré.

Le principal avantage de cette approche est sa simplicité, tant du point de vue de la méthode de synthèse que du concepteur du circuit. En effet, la méthode de synthèse est simple et rapide, puisqu'il suffit de définir, pour chaque construction

du langage, le morceau de circuit correspondant. La synthèse est alors linéaire avec la taille du programme décrivant le circuit. Du point de vue du concepteur, il est relativement facile de deviner ce que donnera le circuit synthétisé, puisqu'il aura une structure similaire au programme. Cependant, l'optimisation est très limitée, puisqu'on ne peut agir qu'au niveau des constructions du langage, donc à un niveau de granularité très grossier.

Parmi les techniques de synthèse de circuits asynchrones, on peut citer l'environnement TIDE, développé par Handshake solution (anciennement Tangram, par Philips) et Balsa, développé à l'Université de Manchester.

2.1 TIDE (Handshake Solutions)

Anciennement un département de Philips Semiconductor, Handshake solution développe et commercialise l'environnement TIDE [15]. Cet environnement cible la conception de circuits asynchrones micropipeline. Il se base sur le langage Haste (anciennement Tangram).

Haste est un langage basé sur le CSP [16] ; il modélise des processus communiquants. C'est un langage de haut niveau, qui manipule des types complexes ; il possède des opérateurs arithmétiques, et des structures de contrôle classiques (`if...then...else`, `for`, etc.). Il comporte un pré-processeur, qui permet d'inclure des fichiers, d'utiliser des options de compilation, de définir des macros, etc.

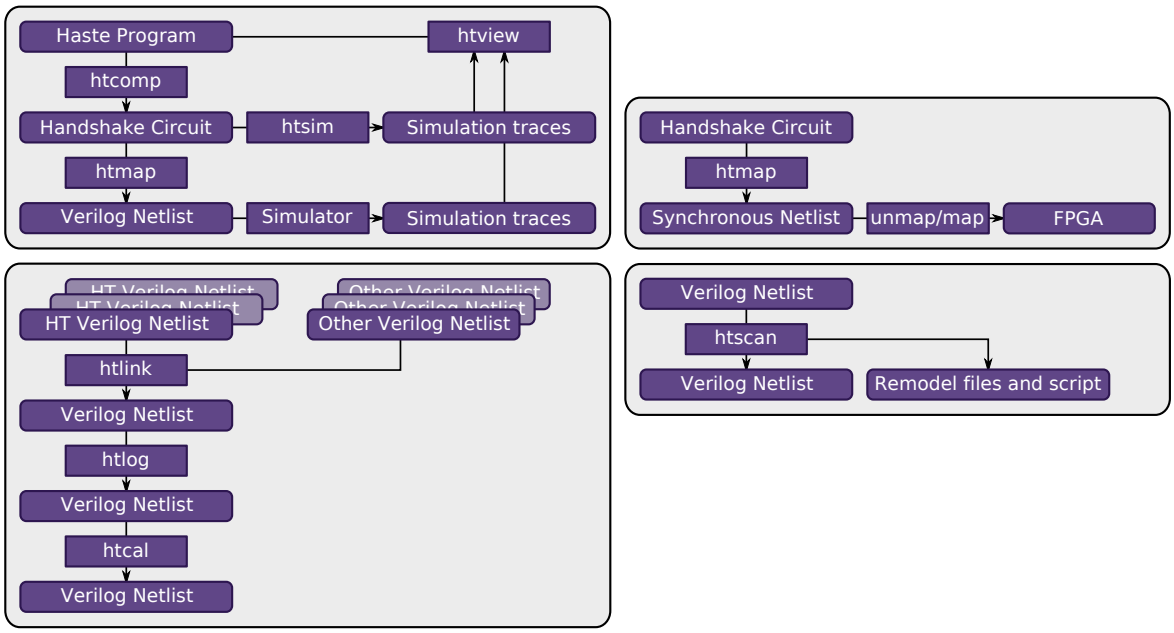


FIG. 2.1 – Le flot de conception TIDE, de Handshake Solutions, basé sur le langage Haste.

La méthode de synthèse utilise un format intermédiaire, qui est un circuit «poignée de mains», basé sur les processus communiquants et les canaux de communications. Le protocole de communications utilisé est un protocole deux ou quatre phases. Les données sont codées soit en données groupées (c'est-à-dire qu'elles sont codées en binaire, et un fil de requête est ajouté), soit en multirails (1-parmi-n).

L'étape de génération de ce format intermédiaire à partir du programme Haste est dirigée par la syntaxe. À partir du format intermédiaire, le circuit est généré grâce à une bibliothèque de circuits «poignée de mains», ainsi qu'un analyseur de performances. Différentes bibliothèques sont utilisées, ce qui permet de cibler différents types de circuits et protocoles (quatre phases ou deux phases, données groupées ou un parmi n), et différentes technologies (cellules standard CMOS, FPGA).

Cette méthode de synthèse est robuste, stable et éprouvée : Philips est l'un des précurseurs des circuits asynchrones. C'est la seule méthode de synthèse de circuits asynchrones actuellement commercialisée.

Cependant, le fait que la synthèse soit dirigée par la syntaxe fait que les circuits générés sont peu optimisés. L'optimisation ne peut se faire qu'au niveau du programme Haste, en modifiant le programme pour utiliser des structures plus efficaces. La granularité est trop grossière pour pouvoir optimiser convenablement les circuits.

2.2 Balsa

L'université de Manchester développe Balsa [17, 18], un système de synthèse de circuits asynchrones, quasi insensibles aux délais ou données groupées. Balsa adopte une approche de la compilation guidée par la syntaxe : les structures du langage sont mappées directement sur des composants communiquants, interconnectés par des canaux et un protocole poignée de mains.

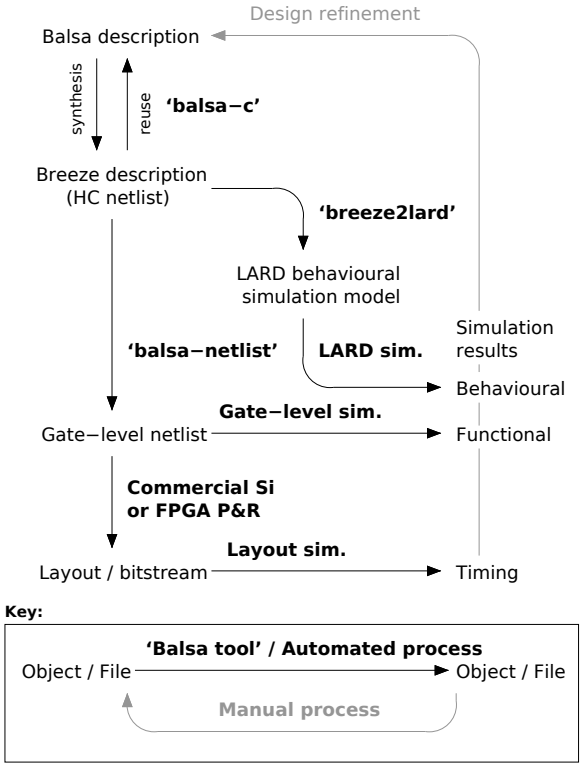


FIG. 2.2 – Le flot de conception de Balsa.

Le langage Balsa est un langage de programmation parallèle : il se base sur la communication synchronisée des canaux et le style de description parallèle de CSP.

Le format intermédiaire utilisé dans le flot Balsa est le format Breeze, qui définit le réseau de circuits communiquant par un protocole poignée de mains. C'est autour de ce format que s'articule l'ensemble d'outils qui forment le flot de conception, présenté figure 2.2.

3 Synthèse par compilation

La synthèse par compilation traduit un programme en circuit par une série de transformations, tout en préservant la sémantique du programme. Cette approche permet d'optimiser les circuits de manière bien plus efficace qu'une approche dirigée par la syntaxe. En effet, le programme est transformé en une succession de descriptions différentes, qui sont choisies pour être adaptées aux optimisations que l'on souhaite effectuer.

En contrepartie de l'efficacité en terme de circuit généré, cette approche est beaucoup plus complexe qu'une méthode dirigée par la syntaxe. En effet, chaque étape de transformation, en plus d'être adaptée pour correctement optimiser le circuit, doit préserver sa sémantique, ce qui demande un soin tout particulier. Une telle approche est nécessaire si l'on veut être capable de générer des circuits suffisamment optimaux pour être capables de rivaliser avec leurs équivalents synchrones.

La première méthode à utiliser une approche par compilation est celle de Caltech, CAST, développée par Alain Martin. On peut également citer TAST, le projet dans lequel s'intègre ce travail de thèse.

3.1 CAST

La méthode développée à Caltech repose sur une description haut niveau du circuit, sous forme de processus communiquants [19, 20]. Le langage de description, CHP [21], est basé sur le langage CSP. Cette modélisation garantit la préservation de la propriété d'insensibilité aux délais pendant tout le flot de synthèse : les processus communiquent entre eux sans jamais faire d'hypothèses concernant la propagation des signaux le long des canaux.

Le principe de la méthode repose sur la modélisation comportementale de chaque processus, et sur des raffinement successifs de cette modélisation à mesure que l'on abaisse le niveau d'abstraction. Au plus haut niveau, le protocole est implicite, et seules les actions de communication (lecture, écriture) sur les canaux apparaissent. Le développement du code permet de faire intervenir différents types de protocole, codage, conventions (choix du processus actif sur un canal, séquencement des communications). Au plus bas niveau, on obtient une description explicite de tous les signaux.

Chaque étape de développement préserve la sémantique du circuit. Mais ces étapes sont complexes, et la plupart sont effectuées manuellement.

D'abord, les processus sont décomposés, afin d'obtenir une collection de processus simples.

Puis l'expansion des communications est effectuée : chaque canal est remplacé par les signaux qu'il contient, en fonction du codage choisi, et chaque action de communication est remplacée par la séquence de transitions correspondante, suivant le protocole utilisé.

Ensuite, à partir des séquences de transitions, sont calculées des règles de productions, qui spécifient le comportement des signaux internes de manière à ce qu'ils respectent ces séquences de transitions. Les règles de productions sont modifiées pour êtres rendues symétriques lorsque ceci est possible, afin de faciliter l'étape suivante.

Enfin, les règles de productions sont regroupées et associées pour former des portes logiques. Le circuit obtenu est spécifié au niveau des transistors. En effet, n'importe quelles règles de production peuvent être obtenues, et il n'est donc pas possible d'utiliser une bibliothèque de cellules, qui ne pourrait contenir toutes les cellules possibles.

3.2 TAST (ancienne version)

TAST est un outil de conception et de synthèse de circuits asynchrones quasi-insensibles aux délais [22]. C'est dans le cadre du développement de cet outil que s'est effectué ce travail de thèse. Cette section présente la méthode de synthèse que TAST implémentait avant ce travail de thèse.

Le langage de description est le CHP, comme dans CAST (basé sur CSP et les processus communiquants). Le flot de synthèse s'articule autour d'un format intermédiaire, qui est un réseau de Pétri dont les places sont étiquetées par des graphes flot de données (DFG). La figure 2.3 montre le flot de conception de cette version de TAST.

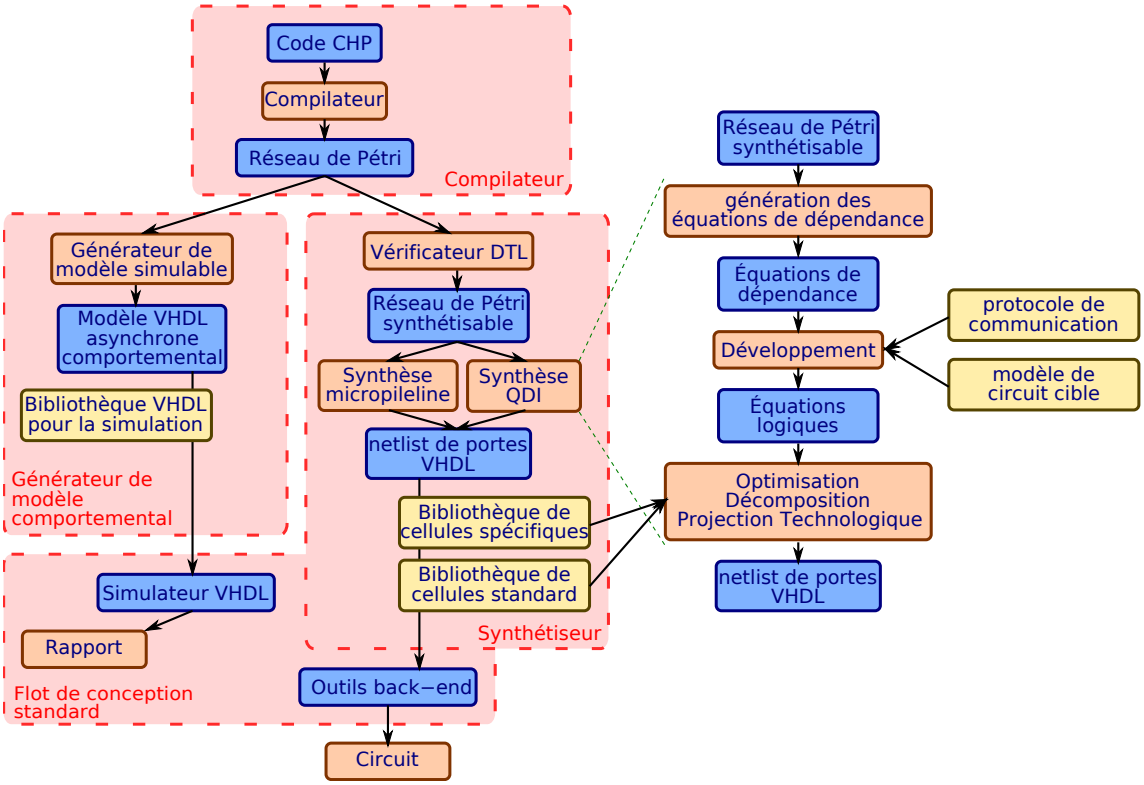


FIG. 2.3 – le Flot de conception de l'ancienne version de TAST

De la même façon qu'il existe en synchrone des règles RTL («Register Transfer Level»), spécifiant si un programme VHDL ou verilog est synthétisable, TAST

définit des règles DTL («Data Transfer Level»). Ces règles identifient les mémoires fonctionelles, et restreignent leur usage. Si un processus ne respecte pas ces règles, il est décomposé en un ensemble de processus qui les respectent.

Une fois que le réseau de Pétri a été rendu DTL, il est synthétisé. Pour cela, il est tout d'abord transformé en une forme intermédiaire, qui est un ensemble d'équations de dépendances.

Les équations de dépendance peuvent être vues comme une fonction logique généralisée, qui contient toutes les informations nécessaires pour synthétiser le processus, de manière indépendante du protocole, pour chaque canal de sortie :

- la validité des canaux d'entrées dont dépend la sortie,
- les conditions qui gardent l'émission d'une donnée sur la sortie,
- la disponibilité du canal de sortie,
- la fonction logique de calcul de la valeur à émettre sur la sortie, en fonction des entrées.

Ces équations sont générées à partir du réseau de Pétri de manière locale : un ensemble de règles spécifie quelle équation générer pour chaque motif possible du réseau de Pétri.

Les équations de dépendance sont ensuite optimisées, afin de supprimer les redondances, par un ensemble de règles simples. Puis elles sont développées afin d'inclure le protocole, qui peut être séquentiel ou WCHB. Enfin, les équations logiques ainsi obtenues sont transformées en une netlist de portes «de base» (en incluant les portes de Muller).

TAST est l'un des premiers outils de synthèse de circuits asynchrones automatique. Le flot est automatisé depuis le langage de description comportementale jusqu'à la netlist. Il ne nécessite l'intervention de l'utilisateur que pour lancer les différentes étapes de la synthèse, et vérifier que tout se passe bien. En particulier, le chemin d'acquittement est calculé automatiquement et de manière totalement transparente.

De plus, on peut utiliser les outils de placement-routage classiques du monde synchrone sur la netlist obtenue après synthèse. En effet, puisque la netlist est quasi insensible aux délais, quelle que soit la manière dont le placement et le routage sont faits, elle fonctionnera. La seule contrainte à respecter est celle des fourches isochrones, qui n'est pas un problème en pratique pour les circuits générés. En effet, il faudrait que la différence de temps de propagation entre deux branches du circuit soit très importante pour que le circuit ne fonctionne pas, et il est facile de contraindre l'outil de placement-routage pour que ceci n'arrive pas.

Cependant, cette version est encore loin d'être comparable à un outil de synthèse de circuits synchrones, et il reste beaucoup d'améliorations à effectuer. C'est le but de cette thèse d'y contribuer.

Tout d'abord, les circuits synthétisés sont sous-optimaux : seules les optimisations naïves sont effectuées. De plus, la structure du circuit généré est encore proche de celle du programme CHP d'où l'on vient : la structure des choix et des gardes est préservée à travers les transformations en réseau de Pétri puis en équations de dépendances, puisque une place du réseau de Pétri est générée par garde, et de même une équation de dépendance est générée par garde. Or, le CHP est un langage qui ne permet pas de manipuler directement des opérations sur des types complexes. Ainsi, la structure de choix et de gardes est très souvent utilisée dans un programme CHP,

de manière imbriquée. Il serait essentiel de s'affranchir de cette structure imbriquée pour générer des circuits vraiment optimisés.

De plus, l'étape de décomposition n'est pas automatisée. Cette étape est pourtant nécessaire ; en effet, les circuits QDI ont tendance à comporter des portes logiques à beaucoup d'entrées, lorsqu'ils sont synthétisés sous forme de portes génériques (autant d'entrées que de canaux d'entrée du circuit). Or, le nombre d'entrées disponible sur les portes des bibliothèques de cellules est limité, et généralement faible : pas plus de 4. Il faut donc décomposer les portes pour que le circuit puisse être projeté sur la bibliothèque de cellules considérée. Cependant, la décomposition des portes de Muller n'est pas simple : si l'on décompose une porte de Muller sans faire attention au reste du circuit, on peut perdre la propriété de quasi-insensibilité aux délais.

Cette version de TAST ne sait pas décomposer automatiquement les portes de Muller. Lorsque l'on rencontre un circuit où il est nécessaire de les décomposer, il faut effectuer la décomposition directement au niveau du programme CHP.

4 Autres méthodes

4.1 Petrify

Petrify [23] synthétise des circuits Indépendants de la Vitesse (SI), à partir d'une spécification sous forme de graphe de transitions de signaux (STG). Ce modèle, basé sur le réseau de Pétri, est une re-formalisation du diagramme temporel, qui spécifie les relations de causalité entre les transitions. Il permet de modéliser la concurrence et une version limitée de choix entre les entrées.

Pour calculer la fonction logique de chaque sortie du circuit, qui est nécessaire pour la synthèse, l'outil doit d'abord calculer l'espace des états atteignables. Mais le nombre d'états de cet espace augmente exponentiellement avec le nombre de signaux du STG. C'est le principal point faible de cette méthode, qui se limite aux petits circuits.

De plus, un problème d'ambiguïté peut se poser lors du calcul des fonctions de signaux : des états différents peuvent avoir le même codage. Dans ce cas, le circuit ne respecte pas la propriété d'État de Codage Complet (CSC, Complete State Coding), et il faut ajouter des variables d'état supplémentaires pour la respecter.

Les équations logiques des sorties sont ensuite calculées, pour les fonctions de charge (*set*) et décharge (*reset*). À partir de ces équations, le circuit est synthétisé en utilisant une bibliothèque de cellules complexes.

Enfin, le circuit est modifié pour pouvoir être initialisé, via l'utilisation d'un signal de Reset.

L'outil de synthèse Petrify automatise cette méthode. Il prend en entrée un STG décrivant le circuit, qui peut être saisi sous forme de texte ou sous forme graphique. La description de circuit sous forme de STG est ardue et sujette aux erreurs ; ceci constitue une limitation de la méthode. De plus, l'étape d'exploration de l'espace d'états limite la complexité des circuits pouvant être réalisés (environ 20 signaux au maximum.

4.2 minimalist

Cette méthode, proposée par Steven Nowick, traite la conception de contrôleurs asynchrones fonctionnant en mode rafale. Elle génère des circuits de Huffman. Elle se base sur une spécification de machine à états. Du point de vue du calcul, cette spécification se base sur les états : à chaque état, la machine peut recevoir des entrées, génère des sorties, et avance jusqu'à l'état suivant. Cette spécification est plus concise que le STG, notamment lorsque la concurrence du système est importante.

Pour que le contrôleur fonctionne correctement, des contraintes sur la spécification sont définies. Tout d'abord, une seule entrée est autorisée à changer dans un état («Single Input Change»). De plus, chaque état a un point d'entrée unique, ce qui simplifie l'optimisation et garantit une logique sans aléa. Pour finir, tout changement d'état nécessite un changement sur une entrée, ce qui signifie que le système reste stable tant qu'aucune entrée ne change.

S. Nowick a également proposé une méthode de synthèse basée sur une machine à états synchronisée localement. Cette machine est une machine de Huffman à laquelle on a ajouté une unité d'auto-synchronisation, qui fonctionne comme une horloge locale.

La première étape de la méthode de synthèse consiste à générer une table de transitions pour les sorties et les prochains états. L'étape suivante optimise la table pour réduire le nombre d'états, et choisit un codage des états. La dernière étape génère les fonctions booléennes pour l'horloge, chaque sortie et chaque variable d'état.

4.3 NCL : Null Convention Logic

Cette logique propriétaire [24–26], brevetée par Thesus Logic, se base sur le codage trois états, qui est une représentation du codage 1-parmi-2. Contrairement à la logique booléenne, où chaque fil ne peut avoir que deux valeurs possibles, 0 et 1, dans la logique NCL3 chaque fil peut avoir trois valeurs possibles : 0, 1 et *Null*. La figure 2.4 montre les tables de vérité des opérateurs de base de la logique NCL3 : si l'une des entrées de l'opérateur est *Null*, la sortie est *Null*, sinon la sortie est la même que pour l'opérateur booléen correspondant.

	0	1	N
0	0	0	N
1	0	1	N
N	N	N	N

(a) AND

	0	1	N
0	0	1	N
1	1	1	N
N	N	N	N

(b) OR

0	1
1	0
N	N

(c) NOT

FIG. 2.4 – Tableau de Karnaugh en logique NCL. La logique NCL comporte trois états : 0, 1 et N (*Null*).

La philosophie de la méthode NCL est d'exploiter au maximum les outils existants, qui ont été développés pour synthétiser des circuits synchrones. Ainsi, la logique 3 états permet d'utiliser les circuits combinatoires synchrones, en remplaçant les portes logiques booléennes par les portes logiques NCL3 équivalentes. Le circuit NCL3 obtenu est sans aléa. Seul le chemin de données est synthétisé ainsi, la gestion des acquittements et de la validité des sorties étant faite manuellement.

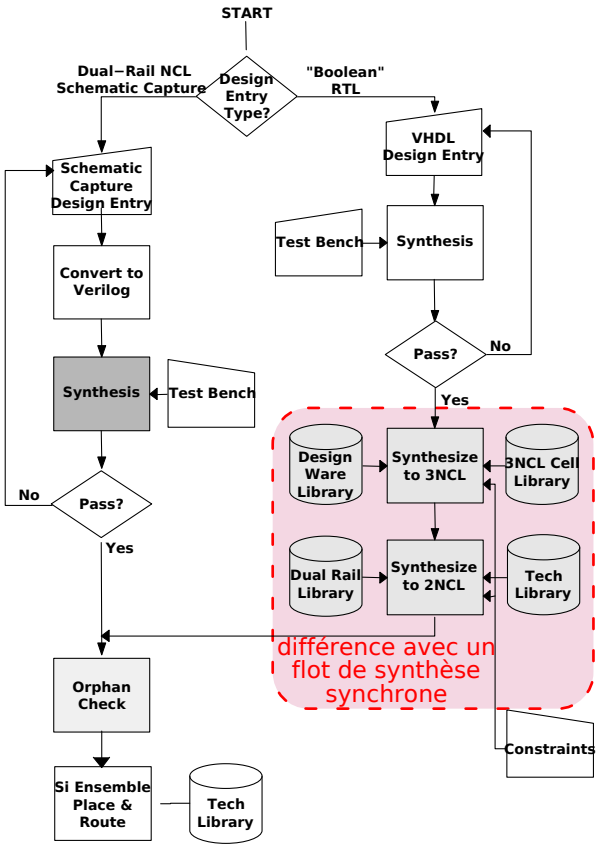


FIG. 2.5 – Le flot de conception NCL

La synthèse est réalisée en deux étapes :

- Le circuit est tout d'abord synthétisé à partir d'une description VHDL par un outil de synthèse commercial, en n'utilisant que des portes de base, qui existent en logique NCL3 (*AND*, *OR*, etc).
- Puis, le circuit NCL3 est transformé en un circuit en logique booléenne classique. Pour cela, chaque fil NCL3 est remplacé par deux fils booléens, en suivant le codage 1-parmi-2, et chaque porte NCL3 est mappée sur un ensemble de portes booléennes.

Le principal intérêt de cette méthode réside dans l'utilisation d'outils existants dans le processus de synthèse, ainsi que dans la spécification sous forme de VHDL. Réutiliser ce qui existe déjà est une manière de faciliter le passage aux technologies asynchrones pour les concepteurs de l'industrie, qui sont ultra-majoritairement formés à la technologie synchrone.

Cependant, générer les circuits d'acquittements à la main demande beaucoup d'efforts. Ceci reste un gros point noir de la méthode.

Chapitre 3

Modélisation de circuits QDI

1 Introduction

Les circuits QDI sont intrinsèquement différents des circuits synchrones, dans leur comportement. Ils sont sensibles aux transitions, qui jouent un rôle particulièrement important puisqu'elles permettent de faire avancer le protocole poignée de mains, alors que les circuits synchrones ne sont sensibles qu'à l'état logique. Il est donc naturel que les modèles pertinents pour décrire des circuits QDI et des circuits synchrones soient différents.

On distingue deux types de modèles de circuits : les modèles comportementaux, qui spécifient le comportement que le circuit doit satisfaire, et les modèles structurels, qui spécifient comment le circuit est implémenté. Les différents modèles de circuit sont autant de vues distinctes d'un même circuit, à des niveaux d'abstraction différents. Un modèle comportemental est plus haut niveau qu'un modèle structurel.

La vérification de la quasi-insensibilité aux délais est un problème difficile [13]. L'approche choisie dans ce travail de thèse pour s'affranchir de ce problème se base sur un modèle de description de circuits QDI sous forme de MDD (diagramme de décision multi-valué), qui spécifie le circuit que l'on souhaite synthétiser. Il permet de générer directement un circuit QDI, sans avoir à faire de vérification de la propriété de quasi insensibilité aux délais : les circuits générés à partir du modèle satisfont intrinsèquement cette propriété. Toutes les optimisations sont faites sur le modèle, ainsi, il n'y a aucune vérification de quasi-insensibilité aux délais à effectuer sur le circuit généré. La preuve de quasi insensibilité aux délais des circuits générés est donnée au chapitre 5.

Ainsi, au plus haut niveau, on modélise le circuit par un programme CHP, c'est-à-dire un ensemble de processus communiquants. Le programme CHP est raffiné pour satisfaire aux règles DTL, et ainsi être synthétisable. Puis, il est transformé en un autre modèle, le modèle de MDD (diagrammes de décision multi-valués), qui spécifie le circuit à un plus bas niveau d'abstraction.

Le modèle de MDD, qui est toujours comportemental, est enfin transformé en une netlist de portes, c'est-à-dire en un modèle structurel. Cette transformation, qui est le cœur de la synthèse, fait l'objet du chapitre 4. De plus, sous certaines hypothèses concernant le MDD, j'ai démontré que cette synthèse génère des circuits respectant la propriété de quasi-insensibilité aux délais. Cette preuve fait l'objet du

chapitre 5.

2 Modèles comportementaux

2.1 Comportement d'un circuit QDI

Avant de présenter les différents modèles comportementaux de circuit QDI, on présente ici ce que l'on cherche à modéliser, c'est-à-dire ce qui définit le comportement que le circuit doit adopter, en fonction des requêtes sur ses entrées et de l'acquittement de ses sorties. Pour cela, on définit les notions de cycle de calcul, et de consommation et de production d'une requête.

Cycle de calcul

On modélise le comportement d'un circuit de manière cyclique :
- Le circuit commence par lire au moins une entrée.
- Puis il émet au moins une sortie, en fonction des entrées lues, et éventuellement de variables internes. C'est durant cette phase qu'a lieu le calcul.
- Enfin, il termine le protocole quatre phases sur certaines entrées lues,
- et sur les sorties qu'il a émises.
- Le circuit est alors prêt pour recommencer le cycle.

Ce cycle est appelé cycle de calcul. Il n'est pas forcément aussi simple qu'il y parait. En effet, à chaque cycle, le circuit n'acquitte pas forcément toutes ses entrées, et n'émet pas forcément une valeur sur toutes ses sorties.

Consommation d'une entrée

Un circuit peut tout à fait lire une entrée durant un cycle de calcul, et ne l'acquitter que durant un cycle de calcul ultérieur. On dit alors que le circuit sonde le canal d'entrée. La donnée sur le canal d'entrée reste alors disponible pour les cycles de calcul suivants, jusqu'à ce que le circuit l'acquitte. On dit alors que la donnée a été consommée : elle n'est plus disponible sur le canal, puisque le circuit qui l'avait émise peut la remettre à zéro. La figure 3.1 illustre la notion de consommation d'une entrée par un *et logique* «paresseux» (c'est-à-dire qui n'évalue que ce qui est nécessaire) : lorsque la première entrée vaut 0, on connaît la valeur à émettre sans avoir à lire la seconde entrée, qui n'est pas consommée.

```
A?a;@[  a='0' => S!0; break
        a='1' => B?b;
                 S!b; break
]
```

(a) code CHP

- lire A
- si $A = 0$, émettre 0 sur S.
- si $A = 1$:
  - lire B
  - émettre la valeur lue de B sur S.

(b) comportement

FIG. 3.1 – exemple de circuit qui ne consomme pas toutes ses entrées à chaque cycle de calcul : *et logique* «paresseux». Si $A = 0$, on émet S, sinon on émet B.

On pourrait décider de toujours consommer la seconde entrée quelle que soit la valeur de la première. Ceci donnerait un circuit différent, alors que l'équation logique

de la sortie serait la même. L'équation logique des sorties n'est donc pas suffisante pour modéliser un circuit QDI. Cette remarque est très importante pour le modèle de circuit QDI que l'on veut définir : en plus de l'équation logique des sorties, il faut que le modèle spécifie si une entrée doit ou non être consommée (c'est-à-dire acquittée).

2.2 Processus communicants : Le langage CHP

Le modèle de plus haut niveau présenté ici est le modèle comportemental de processus communicants. Il est décrit à l'aide du langage CHP ; c'est notre langage de conception. Ce langage est basé sur CSP, et est donc constitué de processus communiquants qui s'exécutent en parallèle. Il intègre explicitement le concept de canal de communication, ce qui le rend parfaitement approprié pour modéliser des circuits asynchrones. La figure 3.2 présente un exemple de programme CHP.

Processus Communicants

Un processus est défini comme une suite d'instructions, qui s'exécutent indépendemment du reste du programme. Ainsi, un programme CHP est constitué d'un ensemble de processus s'exécutant en parallèle.

Les processus peuvent communiquer entre eux, à l'aide de canaux de communication. Un canal de communication relie un processus émetteur à un processus récepteur. Le découpage du programme CHP en un ensemble de processus interconnectés par des canaux de communications lui donne une structure similaire à celle du circuit asynchrone que l'on souhaite modéliser.

```
Op?i; @[ i='0' => A?a, B?b; S!a and b; break
         i='1' => A?a, B?b; S!a  or b; break
         i='2' => A?a, B?b; S!a xor b; break
         i='3' => A?a, B?b;
            @[ a='0' and b='0' => Cout!'0',
                     [Cin?c; S!c]; break
               a='1' xor b='1' => Cin?c;
                     @[ c='0' => Cout!'0', S!'1'; break
                        c='1' => Cout!'1', S!'0'; break
                      ]; break
               a='1' and b='1' => Cout!'1',
                     [Cin?c; S!not c]; break
             ]; break
      ]; loop
```

FIG. 3.2 – Exemple de programme CHP, modélisant une ALU. L'entrée *Op* est lue, et, selon sa valeur, l'opération spécifiée est effectuée, en lisant les entrées nécessaires.

Opérateurs

La suite d'instructions que le processus doit exécuter est définie à l'aide d'un petit nombre d'opérateurs.

Accès à un canal

Les opérateurs de base sont ceux qui permettent au processus d'accéder à ses canaux, pour les lire ou les écrire. En effet, le comportement du processus est défini par les actions qu'il effectue sur ses canaux.

Un processus CHP peut effectuer trois actions sur un canal : lire une donnée, si c'est un canal d'entrée, émettre une donnée, si c'est un canal de sortie, et enfin sonder le canal (s'il est passif, c'est-à-dire un canal d'entrée en mode pull ou un canal de sortie en mode push). La lecture est représentée par le caractère '`?`', l'écriture par le caractère '`!`' et la sonde par le caractère '`#`'.

Contrairement aux actions de lecture et écriture sur un canal, qui font avancer le protocole quatre phase, sonder une canal n'a aucun effet sur le protocole quatre phase. Cette action consiste à tester dans quelle phase du protocole se trouve le canal (c'est-à-dire si une donnée est disponible ou non), et, si une donnée est disponible, tester sa valeur.

Opérateurs séquentiel et parallèle

En CHP, le parallélisme entre les instructions est explicite. Pour cela, deux opérateurs permettent d'enchaîner des instructions : l'opérateur séquentiel, qui indique que la seconde ne commencera que lorsque la première aura terminé, et l'opérateur parallèle, qui indique que les deux instructions s'exécutent en concurrence. L'opérateur séquentiel est représenté par le caractère '`;`' et l'opérateur parallèle par le caractère '`,`'.

Boucles, choix et commandes gardées

Le CHP est un langage impératif. Des opérateurs de boucles et de choix permettent de contrôler le déroulement du programme. Un choix est composé d'un ensemble de gardes (c'est—à-dire de conditions, sous la forme de fonction booléenne) ; à chaque garde est associée un bloc d'instructions : lorsque le programme exécute le choix, il doit exécuter le bloc d'instructions correspondant à la garde qui est vraie.

Un choix peut être déterministe (représenté par le caractère '`@`') ou indéterministe (caractères '`@@`'). Si le choix est déterministe, les gardes doivent être mutuellement exclusives, c'est-à-dire que le programmeur garantit qu'une seule d'entre elles peut être vraie à un instant donné. S'il est indéterministe, plusieurs gardes peuvent être vraies, et c'est au programme de choisir l'une d'entre elles et d'exécuter le bloc d'instructions correspondantes.

Type de donnée présente sur un canal

Le CHP utilise le type le plus général pour les données communiquées sur un canal : la donnée est un nombre, codé par un nombre fixé de chiffres, dans une base fixée. Ainsi, le type `MR[4][5]` contient les nombres représentés par 5 chiffres, en base 4 (c'est-à-dire au plus bas niveau 5 groupes de 4 fils, qui codent chacun l'un des chiffres, en codage 1-parmi-n).

Le type de données de CHP est donc plus général que les types habituellement utilisés, qui se limitent en général à une représentation binaire. La représentation en base quelconque se fait simplement grâce à la représentation 1-parmi-n, qui permet de représenter un chiffre en base n. Le principal intérêt d'utiliser une base non binaire est la réduction de la consommation dynamique du circuit : plus la base est grande, moins le nombre de chiffres nécessaires est grand, et donc moins le nombre de chemins du circuit s'activant est élevé. Or, dans un circuit QDI, il n'y a des transitions que sur les chemins actifs. Et la consommation dynamique est directement dépendante du nombre de transitions dans le circuit.

2.3 Raffinement DTL

De même qu'il existe des règles RTL (Register Transfer Level) pour les circuits synchrones, des règles DTL (Data Transfer Level) sont définies pour les circuits asynchrones [22, 27]. Un programme DTL indique comment les données sont transférées entre les processus. Les règles DTL sont des restrictions dans l'écriture du programme décrivant le circuit, qui garantissent que ce programme est synthétisable. Ce modèle est donc un raffinement du modèle des processus communicants, qui n'est pas forcément synthétisable.

Si un programme spécifiant un circuit asynchrone n'est pas conforme aux règles DTL, il peut être transformé par décomposition [28–30] en un ensemble de processus conformes à la spécification DTL, et donc synthétisables.

Formes synthétisables

Tout comme pour les règles syntaxiques RTL, ce sont sur les éléments de mémorisation du circuit que portent les règles DTL. Dans la conception des circuits asynchrones, une mémoire peut apparaître à différents niveaux :

- La mémoire peut être indiquée au niveau du langage : une mémoire est exigée dès qu'un calcul utilise une variable calculée dans une étape précédente. On parle d'une *mémoire fonctionnelle*.
- Une mémoire apparaît également à chaque fois que le circuit asynchrone implémente un rendez-vous entre les signaux de données : il y a mémorisation du signal interne généré, qui doit rester valide tant que les données en question n'ont pas été remises à zéro. On parle de *mémoire de données*.
- Une mémoire est nécessaire pour certaines implémentations du protocole de communication poignée de mains, car le séquencement des actions de communication entre l'entrée et la sortie est modifié. C'est le cas des protocoles WCHB, PCHB et PCFB. On parle de *mémoire de protocole*.

Les mémoires de données sont implémentées directement dans les cellules logiques de la bibliothèque cible, comme les portes de Muller. Les mémoires de protocole, elles, sont traitées lors de l'implémentation du protocole, durant la synthèse. Ainsi, seules les mémoires fonctionnelles sont considérées dans la définition des règles DTL.

Un circuit asynchrone peut être vu comme une fonction qui réagit sous certaines conditions, en émettant la sortie en fonction des entrées. Deux cas peuvent être envisagés :

- Le circuit produit un résultat qui ne dépend que que la valeur courante des entrées : c'est un circuit combinatoire.

– Le circuit produit un résultat qui dépend non seulement de la valeur courante des entrées, mais aussi de son évolution passée : c'est un circuit séquentiel. Le circuit évolue au cours du temps en effectuant des transitions d'un état à un autre. Il peut être modélisé par une machine à états finie (FSM), représentée figure 3.3.

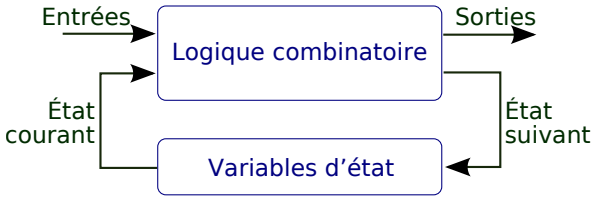


FIG. 3.3 – modèle d'un circuit asynchrone sous forme de machine à états finie

Ce sont les circuits séquentiels qui comportent des éléments de mémorisation, et qui nous intéressent ici. Encore une fois, il y a plusieurs types de mémoire fonctionnelles possibles :

Mémoire causée par une variable locale .

Une variable déclarée, affectée et lue peut représenter une mémoire fonctionnelle, lorsque sa valeur dépend de l'état précédent du processus (par exemple lorsqu'elle a été affectée lors du précédent cycle de calcul). On parle alors de *variable mémorisante*. Par opposition, une *variable intermédiaire*, qui est affectée puis lue dans le même cycle de calcul, ne crée pas de mémoire fonctionnelle.

Pour être combinatoire, un circuit ne doit pas avoir de variable mémorisante.

Mémoire causée par l'opérateur séquentiel .

L'utilisation de l'opérateur séquentiel (symbolisé par le caractère ' ;' en CHP) dans un programme peut créer des états, et donc faire apparaître des mémoires fonctionnelles.

Par exemple, le programme CHP `[E?x;E?x;S!x;loop]`, qui lit deux fois successivement le canal d'entrée E, et ne propage que la seconde valeur lue sur la sortie S, est séquentiel. Cependant, tous les opérateurs séquentiels ne produisent pas de mémoire fonctionnelle. Dans l'exemple précédent, seul l'opérateur séquentiel séparant les deux lectures produit une mémoire fonctionnelle. Les autres opérateurs séquentiels s'implémentent simplement, car il existe une dépendance de donnée entre les deux instructions.

Initialisation du circuit .

Dans la modélisation d'un système asynchrone avec des processus concurrents communicants, deux types d'initialisation sont envisagées : l'initialisation des variables, et l'initialisation des canaux de communication. Ces deux types d'initialisation produisent des mémoires fonctionnelles.

L'initialisation des canaux de communication permet d'émettre une première valeur sur un canal, avant de passer à un «régime permanent» où le circuit émet des valeurs calculées en fonction des entrées. Ce type d'initialisation peut être implémentée en ajoutant un opérateur d'initialisation sur le canal, par la technique décrite section 3.7 du chapitre 1.

En revanche, l'initialisation des variables n'est pas synthétisable ; elle doit être interdite dans les règles DTL.

Règles DTL

L'analyse des éléments de mémorisation impliqués dans la réalisation d'un circuit asynchrone faite ci-dessus permet de définir un ensemble de règles de modélisation et d'écriture, afin que le circuit réalisé puisse être réalisé par une FSM, selon le modèle présenté figure 3.3. Pour être conforme à l'écriture DTL, un processus doit respecter toutes les règles suivantes :

1. Les variables partagées en parallèle ne sont pas autorisées, sauf dans le cas où elles ne sont accédées qu'en lecture.
2. Les variables doivent être affectées avant d'être lues
3. Les valeurs émises sur les canaux de sortie doivent exclusivement dépendre des valeurs reçues sur les canaux d'entrée.
4. Un même canal ne peut être accédé séquentiellement que pour être initialisé.
5. Un même canal ne peut être accédé de manière concurrente
6. Deux instructions séquentielles doivent exprimer une dépendance de données.

Les règles 1 à 3 restreignent l'utilisation de variables de mémorisation. Une variable doit avoir une valeur avant d'être consultée. Ni variables ni canaux ne conservent leur valeur au cours de leur évolution : ils obtiennent une nouvelle valeur à chaque cycle de calcul.

La restriction sur l'opérateur séquentiel (règles 4 et 6) permet de déduire la relation entre canaux de sortie et canaux d'entrée. La règle 5 garantit qu'il n'y a pas de conflit de données sur un canal : il ne peut pas être consommé dans deux branches du programme à la fois. De même, un canal de sortie ne peut pas émettre deux valeurs à la fois.

2.4 Modèle de Diagramme de Décision Multi-valué (MDD)

Présentation

La structure de MDD est utilisée dans cette thèse pour modéliser les circuits QDI que l'on souhaite synthétiser. Cette structure est utilisée car elle met en valeur la propriété de mutuelle exclusion des différents rails, propriété qui est essentielle pour montrer la quasi-insensibilité aux délais.

Cette structure [1, 2] est une généralisation de la structure de BDD [31, 32] (Binary Decision Diagram), qui est utilisée dans de nombreux outils de synthèse de circuits synchrones [33–35], et naturellement considéré pour la synthèse de circuits asynchrones [36]. La figure **??** montre un exemple de MDD. La structure ressemble à un arbre qui spécifie, pour chaque combinaison de valeurs des variables d'entrée, la valeur que doit prendre la sortie. En fait, ce n'est pas tout à fait un arbre, puisque deux nœuds peuvent avoir le même fils.

On peut noter que, tout comme pour les BDD, cette structure est dépendante du choix d'un ordre des variables : deux MDD spécifiant le même circuit, avec des ordres de variables distincts, auront une structure disctincte, et donc des tailles différentes. Ceci cause un problème pour l'optimisation de cette structure, étant donné que le nombre d'ordres possibles d'un ensemble de n variable est $n!$, c'est-à-dire plus qu'exponentiel.

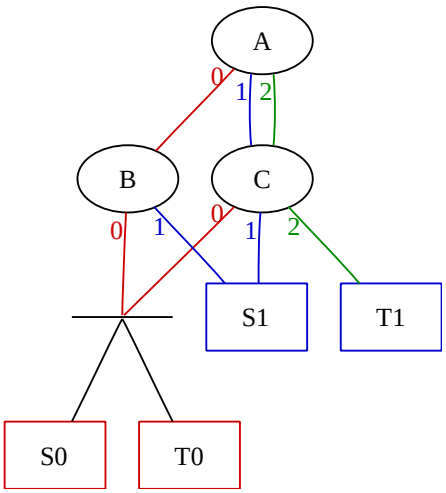


FIG. 3.4 – Un MDD spécifie, pour chaque combinaison des variables d'entrées, la valeur que les sorties doivent prendre.

Cependant, contrairement aux BDDs, la structure de MDD utilisée ici n'est pas totalement libre dans l'ordre de ses variables : il faut respecter les dépendances de lecture des variables.

Pour décrire entièrement un circuit, le modèle doit spécifier la valeur de chaque sortie (et, en même temps, la validité), ainsi que l'acquittement de chaque entrée. Pour cela, le MDD est constitué de plusieurs composantes, qui sont utilisées conjointement, comme l'illustre la figure 3.5(b).

On remarque que dans le MDD, les composantes spécifiant chaque sortie et l'acquittement de chaque entrée ne sont pas indépendantes. En effet, il peut y avoir des sous-expressions communes à ces différentes composantes, qui doivent pouvoir être mises en commun si l'on veut pouvoir optimiser le modèle.

On note aussi la présence de nœuds spéciaux, appelés fourches, qui diffusent un signal sur plusieurs nœuds fils. Les fourches permettent à une branche d'émettre une valeur sur plusieurs sorties à la fois, ce qui est nécessaire pour pouvoir factoriser, c'est-à-dire mettre en commun des branches du mdd. De plus, on considère les signaux d'acquittement comme des données 1-parmi-n, avec $n = 1$. En effet, un signal est une valeur booléenne, qui peut valoir soit 0 (invalide), soit 1 (valide).

Définition formelle

On définit formellement la structure de MDD.

Soient I un ensemble d'entrées, O un ensemble de sorties et T un ensemble de variables intermédiaires : ces trois ensembles sont disjoints. On construit l'ensemble de variables $V = I \cup O \cup T$. À chaque variable $v \in V$ est associé l'ensemble de valeurs qu'elle peut prendre, noté $range(v)$.

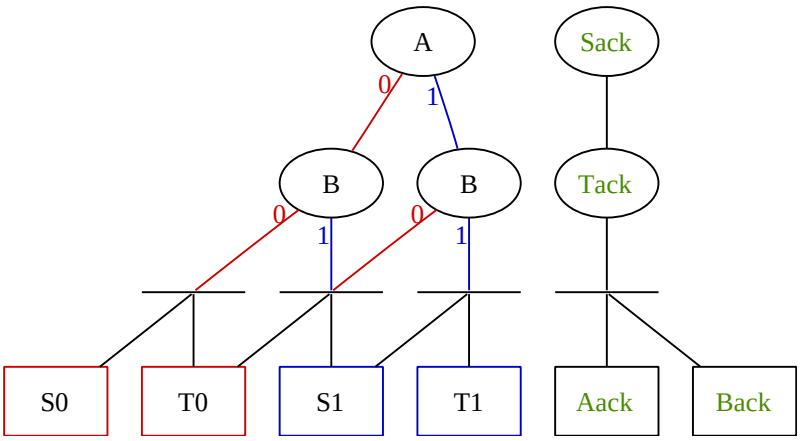
Soit N un ensemble de nœuds et E un ensemble d'arcs de N, c'est-à-dire $E \subset N^2$.

Définition 2.1 *Un MDD (N, E) sur (I, O, T) est un graphe orienté acyclique tel que :*

– Il y a trois types de nœuds : terminal, non-terminal et fourche.

$A?\,a\,,\ \ B?\,b\,;$
$S!\,a\ \ or\ \ b\,,$
$T!\,a\ \ and\ \ b$

(a) code CHP



(b) MDD

FIG. 3.5 – Un MDD décrivant un circuit QDI. Ce circuit a deux entrées et deux sorties, calculant respectivement le *ou logique* et le *et logique* des entrées. Les traits horizontaux représentent des nœuds fourches. Les composantes du MDD spécifiant S et T partagent une partie de la structure, située au-dessus des fourches.

- *Un nœud terminal n n'a pas d'arc sortant. Il est étiqueté par une variable de sortie $v = var(n) \in O \cup T$ et une valeur $val(n) \in range(v)$.*
- *Un nœud non-terminal n est étiqueté par une variable $v = var(n) \in I \cup T$. Il a exactement $|range(v)|$ arcs sortants, un par valeur possible de la variable v. Chaque arc sortant est associé à une valeur $i \in range(v)$, et le nœud vers lequel cet arc se dirige est noté $child(n, i)$.*
- *Une fourche a un nombre quelconque de fils (éventuellement 0) et n'a pas d'étiquette.*

On note $ein(n)$ l'ensemble des arcs entrants du nœud n et $eout(n)$ l'ensemble de ses arcs sortants. On a donc les relations suivantes, si n est un non-terminal et $i \in range(var(n))$: $(n, child(n, i)) \in eout(n)$ et $(n, child(n, i)) \in ein(child(n, i))$.

Définition 2.2 *Un nœud n d'un MDD (N, E) est une racine si et seulement si n n'a pas d'arc entrant, c'est-à-dire si et seulement si $ein(n) = \emptyset$*

Un MDD n'est pas forcément connexe, et peut avoir plusieurs racines.

On définit ensuite les chemins d'un MDD, qui vont d'une racine à une feuille.

Définition 2.3 *Un **tronçon** t d'un MDD $M = (N, E)$ est :*
- *soit un couple (n, i), n étant un non-terminal de M et $i \in range(var(n))$,*
- *soit une fourche de M,*
- *soit un terminal de M.*

Le nœud du tronçon est le non-terminal n si le tronçon est un couple (n, i), et la fourche ou le terminal sinon. Il est noté $node(t)$. Si $node(t)$ est un non-terminal, on note $val(t)$ la valeur considérée dans le tronçon, c'est-à-dire telle que $t = (node(t), val(t))$.

Définition 2.4 *Un nœud $n' \in N$ est un **successeur** d'un tronçon t si et seulement si l'une des deux conditions suivantes est vraie :*
- *t est un couple (n, i) et $n' = child(n, i)$,*

– *t est une fourche et $(t, n') \in E$.*

Un terminal n'a pas de successeur.

Un tronçon t' est un successeur d'un tronçon t si et seulement si $node(t')$ est un successeur du tronçon t.

Définition 2.5 *Un **chemin** d'un MDD M est un ensemble ordonné de tronçons de M $t_0, t_1, \ldots, t_k$ tel que :*

– *$node(t_0)$ est une racine de M*

– *$node(t_k)$ est un terminal de M*

– *pour $0 < i \leq k$, t_i est un successeur de t_{i-1}.*

Comme un terminal n'a pas de successeur, $node(t_k)$ est l'unique terminal dans le chemin.

Contraintes

La structure de MDD définie ci-dessus est très générale, et en l'état permet des aberrations. Par exemple, elle permet d'émettre plusieurs valeurs à la fois sur un même canal de sortie, ce qui est bien entendu interdit.

De même, cette structure permet à un nœud d'être activé par deux signaux qui ne sont pas exclusifs. Ceci est interdit dans un circuit QDI, car le premier signal à arriver causerait une transition en sortie, et le second ne serait pas acquitté. Or on a vu que dans un circuit QDI, toute transition doit être acquittée par une transition en sortie.

Pour éviter ce problème, on définit une propriété de MDD bien formé, que doit respecter tout MDD. Cette propriété permet de garantir que les différents rails d'un même canal de sortie sont mutuellement exclusifs, c'est-à-dire qu'un seul d'entre eux peut être à 1 à un instant donné. Elle fait pour cela l'hypothèse que les rails des canaux d'entrée sont bien eux-mêmes mutuellement exclusifs.

On commence par définir la mutuelle exclusion de deux arcs :

Définition 2.6 *Deux arcs a et b sont **mutuellement exclusifs** si et seulement si ils respectent l'une des conditions suivantes :*

– *a et b sont des arcs distincts sortant d'un même nœud non-terminal ($\exists n \in N, a \neq b$ et $\{a, b\} \subset eout(n)$),*

– *a est un arc sortant d'un nœud n, et b est mutuellement exclusif avec tous les arcs entrants de n ($\exists n \in N, a \in eout(n)$ et $\forall c \in ein(n), b$ et c sont mutuellement exclusifs).*

Un ensemble d'arcs A est mutuellement exclusif si et seulement si les arcs sont mutuellement exclusifs deux à 2, c'est-à-dire si et seulement si $\forall (a, b) \in A^2$, $a \neq b \implies a$ et b sont mutuellement exclusifs.

Deux chemins sont mutuellement exclusifs si leurs terminaux sont mutuellement exclusifs.

Cette définition est récursive : le second point de la définition fait appel à la notion de mutuelle exclusion elle-même. Elle ne peut pas être cyclique (la récursion termine) : en effet, la définition de mutuelle exclusion de deux arcs a et b utilise la mutuelle exclusion de a avec les arcs prédécesseurs de b dans le MDD. Comme le

MDD est acyclique, en itérant la définition on ne peut pas retomber sur la mutuelle exclusivité de a et b.

On définit maintenant un MDD bien formé :

Propriété 2.1 *Un MDD (N, E) est bien formé si et seulement si :*
- *Les arcs entrants d'un nœud sont mutuellement exclusifs : $\forall n \in N, ein(n)$ est mutuellement exclusif.*
- *Les terminaux étiquetés par la même variable sont mutuellement exclusifs : si u et v sont tels $var(u) = var(v)$ et $val(u) \neq val(v)$, alors u et v sont mutuellement exclusifs.*

Ainsi, une seule valeur peut être émise à la fois sur un canal de sortie, puisque les différents terminaux étiquetés par ce canal sont mutuellement exclusifs : un seul peut être activé à la fois. De même, si un nœud peut être activé par plusieurs signaux, ceux-ci sont forcément mutuellement exclusifs, donc un seul l'activera à la fois, causant forcément une transition en sortie.

Sémantique d'un MDD

Un MDD décrit le comportement que doivent avoir les variables et les sorties en fonction des entrées. Pour le moment, la structure du MDD a été définie. Il reste à faire le lien avec le circuit, c'est-à-dire spécifier quel comportement du circuit modélisé correspond à un MDD. C'est ce que l'on appelle la sémantique du MDD.

Le circuit modélisé est QDI, avec un protocole 4 phases. Chaque arc du graphe correspond à un fil dans le circuit. Les arcs seront assimilés au fil correspondant dans la suite.

Les entrées, sorties, et variables sont codées par un code insensible aux délais, qui code la requête avec les données. Elles sont donc soit invalides, soit valides avec une valeur (dans le cas de variables d'acquittement, il y a une seule valeur valide).

La sémantique d'un MDD est définie pour un MDD bien formé. En particulier, les arcs entrants d'un nœud sont mutuellement exclusifs. Le comportement du circuit n'est pas spécifié si plusieurs d'entre eux sont à 1 en même temps.

On définit deux états dans lesquels peuvent se trouver les nœuds et les arcs d'un MDD : activé, et désactivé.
- Les racines sont toujours activées.
- Par défaut, les nœuds qui ne sont pas racines sont désactivés.
- Un nœud qui n'est pas racine est activé si et seulement si l'un de ses arcs entrants est activé.

Lorsqu'un nœud est activé, il peut activer certains arcs sortants, et ainsi certains nœuds successeurs. Soit n un nœud activé.
- Si n est un non-terminal, $var(n)$ est lue. Si $var(v)$ est valide, avec la valeur i, l'arc $(n, child(n, i))$ est activé.
- Si n est une fourche, tous ses arcs sortants sont activés.
- Si n est un terminal, la valeur $val(n)$ est émise sur $var(n)$.

De même, lorsqu'un nœud est désactivé, il peut désactiver certains arcs sortants, et ainsi certains nœuds successeurs. Soit n un nœud désactivé.
- Si n est un non-terminal, $var(n)$ est lue. Si $var(v)$ est invalide, tous les arcs sortants de n sont désactivés.

– Si n est une fourche, tous ses arcs sortants sont désactivés.
– Si n est un terminal, la variable $var(n)$ est remise à zéro.

Les différents terminaux étiquetés par une même sortie sont mutuellement exclusifs, donc un MDD ne peut émettre qu'une unique valeur par sortie à un instant donné. Pour cela, on montre le théorème 2.1 :

Théorème 2.1 *Un terminal n devient activé lorsqu'il existe un chemin ayant pour feuille n dont tous les nœuds sont activés. De même, un terminal n devient désactivé lorsque tous les nœuds de tous les chemins ayant pour feuille n sont désactivés.*

En effet, un terminal n devient activé lorsque l'un de ses parents est activé. Récursivement, on remonte ainsi un chemin, jusqu'à arriver à la racine. De même pour la désactivation.

Comme deux chemins arrivant sur des terminaux d'une même sortie sont mutuellement exclusifs, un seul peut être actif à la fois. Donc un seul terminal concernant cette sortie peut être actif à la fois, et une seule valeur peut être émise.

MDD direct et MDD d'acquittement

Jusqu'à présent, les variables étiquetant les nœuds du MDD étaient quelconques. Mais on veut se rattacher au circuit que l'on souhaite modéliser.

La façon la plus simple de faire est de considérer que l'ensemble des variables d'entrées I est l'ensemble des canaux d'entrée du circuit, et l'ensemble des variables de sortie O est l'ensemble des canaux de sortie du circuit.

Ceci permet de modéliser les chemins de données du circuit, c'est-à-dire les chemins qui calculent les requêtes de sorties et les valeurs à émettre sur ces sorties.

Cependant, on veut aussi modéliser les chemins d'acquittement du circuit. Pour ces chemins, on calcule l'acquittement des canaux d'entrées en fonction des acquittements des canaux de sorties, et éventuellement des valeurs des canaux d'entrée. Il faut donc ajouter dans I les acquittements des canaux de sortie, et dans O les acquittements des canaux d'entrée.

Ainsi, si l'on note $\mathcal{I}$ et $\mathcal{O}$ les ensembles de canaux d'entrées et de sortie du circuit, on a $I = \mathcal{I} \cup (\mathcal{O} \times \{ack\})$ et $O = \mathcal{O} \cup (\mathcal{I} \times \{ack\})$ avec, pour $v \in (\mathcal{I} \cup \mathcal{O}) \times \{ack\}$, $|range(v)| = 1$, puisque les signaux d'acquittement sont considérés comme 1-parmi-1.

3 Modèles structurels

3.1 Structure d'un circuit QDI

Le modèle comportemental de circuit est nécessaire pour spécifier les circuits que l'on veut synthétiser. L'étape de synthèse, à partir de ce modèle comportemental, doit générer un modèle structurel du circuit, sous la forme de netlist de portes. Cette netlist a une structure particulière, liée au comportement du circuit, que l'on détaille ici.

Chaque canal comporte trois éléments : la donnée, la requête et l'acquittement. La donnée et la requête se propagent dans le même sens (de l'émetteur vers le

récepteur), et sont liées grâce au codage insensible aux délais. L'acquittement, lui, se propage dans le sens opposé (du récepteur du canal vers son émetteur).

Ainsi, on peut classer en deux catégories les calculs qu'un circuit doit réaliser :

- Pour chaque canal de sortie, le circuit doit calculer le signal de requête (c'est-à-dire déterminer s'il doit émettre une donnée sur le canal durant ce cycle de calcul), et conjointement la donnée à émettre le cas échéant.
- Pour chaque canal d'entrée, il doit calculer le signal d'acquittement (c'est-à-dire déterminer s'il a consommé la donnée présente sur le canal).

On considère maintenant les chemins de la netlist de portes modélisant le circuit après synthèse. Ces chemins suivent les mêmes contraintes que les calculs, puisqu'ils implémentent ces calculs. Ils se classent donc dans les mêmes catégories, selon qu'ils arrivent sur un canal de sortie ou sur l'acquittement d'un canal d'entrée. On parle de chemins de données et de chemins d'acquittement, faisant référence à ce que le chemin calcule. La figure 3.6 illustre les chemins d'un circuit QDI.

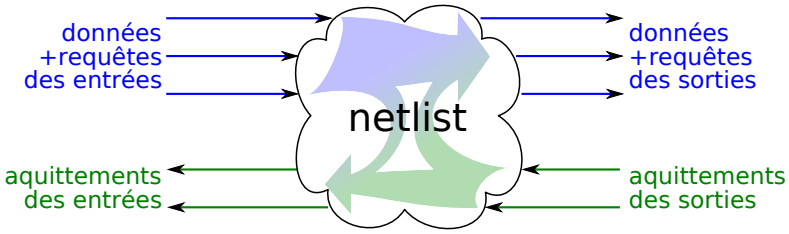


FIG. 3.6 – Un circuit QDI calcule les données et requêtes de ses sorties et les acquittements de ses entrées en fonction des données et requêtes de ses entrées et des acquittements de ses sorties.

3.2 Netlist

Une netlist est une structure, proche du graphe, qui permet de représenter un circuit. La différence se situe au niveau des interconnexions entre les éléments : dans un graphe, les nœuds sont directement reliés entre eux, alors que dans une netlist, les instances contiennent des ports, et ce sont les ports qui sont interconnectés par des nets, comme l'illustre la figure 3.7.

Dans ce manuscrit, on fait le choix de décomposer les fourches, qui sont des hyperarcs, en un ensemble d'arcs ayant la même source. Ces arcs sont appelés nets.

Définition

Formellement, à partir d'un ensemble de ports d'entrée P_i et de ports de sortie P_o, on construit l'ensemble des instances en regroupant les ports en sous-ensembles disjoints de l'ensemble $P_i \cup P_o$. Les nets vont d'un port de sortie à un port d'entrée, donc l'ensemble des nets est inclus dans $P_o \times P_i$.

Définition 3.1 *Une **netlist** est un quadruplet (G, P_i, P_o, N) où :*

- *$G \subset \mathscr{P}(P_i \cup P_o)$ est un ensemble de sous-ensembles non vides disjoints de $P_i \cup P_o$. Les éléments de G sont les **instances**.*

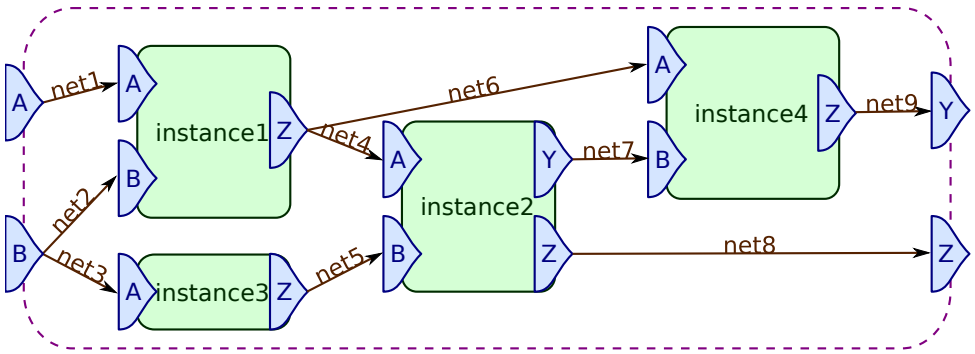


FIG. 3.7 – Une netlist est composée d'instances, qui contiennent des ports d'entrée et des ports de sortie. Les ports de sortie sont reliés à des ports d'entrées via des nets. Les ports qui n'appartiennent à aucune instance sont les ports primaires de la netlist.

- *P_i et P_o sont les ensembles de **ports d'entrées** et de **ports de sortie**. Soit une instance $g \in G$, $g \cap P_i$ est l'ensemble des ports d'entrée de g, et $g \cap P_o$ est l'ensemble de ses ports de sortie.*
- *$N \subset P_o \times P_i$ est un ensemble de couples de ports, appelés **nets**. Un net relie un port de sortie à un port d'entrée.*

On peut remarquer que dans cette définition, tous les ports n'appartiennent pas forcément à une instance. On appelle ports primaires les ports qui n'appartiennent à aucune instance : ce sont les ports de la netlist elle-même.

Définition 3.2 *Les **ports primaires** d'une netlist (G, P_i, P_o, N) sont les éléments de l'ensemble $(P_i \cup P_o) - \cup_{g \in G} g$.*

Instanciation

Dans la définition ci-dessus, les instances sont considérées comme des objets quelconques ; chaque instance est différente. Cependant, comme leur nom l'indique, les instances sont obtenues par instanciation : dans un circuit, on distingue les portes OR, qui sont des instances de la cellule OR de la bibliothèque, les portes de Muller, qui sont des instances de la cellule correspondante, etc.

Pour cela, on étiquette chaque instance par l'objet instancié.

Définition 3.3 *Soit B un ensemble, qui est appelé **bibliothèque**. Soit (G, P_i, P_o, N) une netlist. Pour chaque instance $g \in G$, on note $mod(g) \in B$ l'élément de la bibliothèque dont g est une instance.*

Le contenu de la bibliothèque est très général. Ses éléments identifient les portes logiques disponibles. On peut disposer d'une implémentation de la porte logique, de la fonction logique réalisée, de caractéristiques physiques de la cellule, etc. Tout ceci est géré à un niveau plus bas que ce qui nous intéresse ici ; le contenu de la bibliothèque n'est donc pas spécifié.

Graphes issu d'une netlist

Une netlist a une structure très similaire à un graphe orienté. Tellement similaire que les algorithmes de parcours d'une netlist sont les mêmes que pour un graphe. Pour formaliser ceci, et ne pas avoir à redéfinir ces différents algorithmes, on définit deux graphes issu d'une netlist :

Le **graphe d'instances**

L'ensemble des nœuds de ce graphe est l'ensemble G des instances de la netlist. L'ensemble V des arcs du graphe est défini de la manière suivante : il existe un arc entre deux instances s'il existe un net allant d'un port de sortie de la première instance à un port d'entrée de la seconde.

Définition 3.4 *Le* ***graphe d'instances*** *d'une netlist* (G, P_i, P_o, N) *est le graphe* (G, V) *où* $(g_1, g_2) \in V$ *si et seulement si* $\exists (p_1, p_2) \in N, p_1 \in g_1$ *et* $p_2 \in g_2$.

Le **graphe de ports**, qui est un graphe orienté biparti.

L'ensemble des nœuds de ce graphe est l'ensemble $P_i \cup P_o$ des ports de la netlist. L'ensemble V' des arcs du graphe est défini de la manière suivante : il existe un arc entre un port de sortie et un port d'entrée s'il existe un net reliant ces deux ports. Il existe un arc entre un port d'entrée et un port de sortie si les deux ports appartiennent à la même instance.

Définition 3.5 *Le* ***graphe de ports*** *d'une netlist* (G, P_i, P_o, N) *est le graphe* $(P_i \cup P_o, V')$ *où* $V' \subset P_i \times P_o \cup P_o \times P_i$ *et* $(p1, p2) \in V'$ *si et seulement si* $(p1, p2) \in P_o \times P_i$ *et* $(p1, p2) \in N$ *ou* $(p1, p2) \in P_i \times P_o$ *et* $\exists b \in B, \{p1, p2\} \subset B$.

Ainsi, il est possible d'utiliser les algorithmes classiques de parcours de graphe orienté pour parcourir une netlist. Le graphe d'instances perd l'information des ports de la netlist, information qui peut être utile pour le traitement que l'on souhaite effectuer en parcourant cette netlist. D'un autre côté, le graphe de ports contient plus de nœuds, et il est donc plus coûteux à parcourir.

De plus, le graphe d'instances a une structure similaire à la netlist d'où il est issu. En effet, son ensemble de nœuds est l'ensemble d'instances de la netlist, et il existe une bijection entre l'ensemble des arcs du graphe d'instances et l'ensemble des nets de la netlist. Ainsi, si l'on assimile les arcs du graphe aux nets de la netlist, on peut définir des chemins dans une netlist, qui sont les chemins du graphe d'instances.

Dans toute la suite, on identifiera une netlist et le graphe d'instances correspondant.

3.3 Hiérarchie

Dans tout ce qui précède, on a modélisé le circuit sans hiérarchie : le circuit a des entrées, des sorties, et respecte une spécification. Cependant, quand on veut concevoir un gros circuit, on souhaite découper le problème en plusieurs sous-problèmes : on souhaite spécifier le circuit sous forme d'un ensemble de composants interconnectés.

En fait, ces composants interconnectés forment une netlist : la structure est exactement la même que la netlist de portes ci-dessus. Ce qui change, c'est que les

instances sont différentes. Ce ne sont plus des instances de portes, mais des instances de composants eux-mêmes : les composants sont définis de manière récursive. On parle alors de structure hiérarchique : puisqu'un composant est composé d'instances d'un certain nombre d'autres composants, il existe une hiérarchie entre les composants.

Cette structure hiérarchique a deux intérêts :

- La réutilisation de composants, qui permet de ne spécifier qu'une seule fois une partie du circuit qui sera utilisée à plusieurs endroits. Ceci permet de développer plus rapidement les circuits.
- La division du problème : plutôt qu'avoir une énorme netlist sur laquelle travailler, on a un ensemble de netlists de taille plus réduite, sur lesquelles il est plus rapide d'effectuer des tâches complexes.

La récursivité n'est bien entendu pas infinie (définition cyclique interdite). Ainsi, certaines instances ne sont pas étiquetées par un composant, mais bien par une porte logique, dont on connaît l'équation logique, et donc l'implémentation, comme on le considérait dans ce qui précède.

Il existe alors un type particulier de composant : ceux qui ne contiennent que des instances modélisées par une porte logique. Plus simplement, ce sont les composants pour lesquels il n'y a pas de récursion, c'est-à-dire les feuilles de l'arbre de hiérarchie les composants. Un tel composant correspond à un processus du programme CHP. La figure 3.8 illustre un circuit hiérarchique : le composant `full_adder` est instancié quatre fois dans le composant `adder`, qui effectue une addition sur quatre bits.

Pour simplifier, on interdit les netlists «mixtes», dont certaines instances seraient des composants alors que d'autres seraient des portes logiques. On peut toujours se ramener à un circuit qui respecte cette hypothèse. En effet, si l'on considère une netlist «mixte», il suffit d'enrober toutes les instances modélisées par une porte logique dans un nouveau composant, qui contient uniquement ces portes logiques, et on obtient une netlist qui vérifie l'hypothèse.

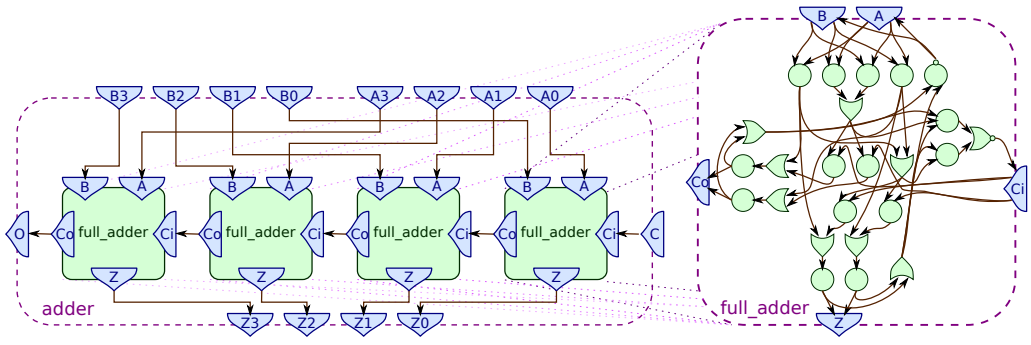


FIG. 3.8 – Exemple de circuit hiérarchique. Le composant `adder` est implémenté par une netlist dans laquelle le composant `full_adder` est instancié 4 fois. Le composant `full_adder`, lui, est implémenté par une netlist de portes logiques (*or*, *nor* et porte de Muller).

Les processus jouent le rôle le plus important du point de vue de la synthèse, car c'est à leur niveau que tout se passe. En effet, la structure hiérarchique est définie par les concepteurs du circuit, et elle est préservée jusqu'à l'étape de mise à plat, qui remplace la structure hiérarchique par une énorme netlist de portes logiques .

Tous les traitements que fait l'outil de synthèse doivent préserver la hiérarchie du circuit.

Dans toute la suite, on considère les circuits à plat, en oubliant la hiérarchie. Faire la synthèse d'un programme hiérarchique revient à faire la synthèse de chaque processus, qui est à plat.

3.4 Règles de productions

Un ensemble de règles de production permet de décrire une netlist de portes logiques qui constituent un circuit. Ce modèle, qui est utilisé dans le flot de conception de CAST, sera utilisé dans le chapitre5 pour modéliser la netlist de manière formelle, afin de prouver formellement la quasi insensibilité aux délais des circuits synthétisés sous forme de netlist.

Dans ce modèle, chaque porte logique est modélisée par une paire de règles de production : l'une des règles spécifie les conditions nécessaires pour que la porte génère une transition montante, et l'autre règle spécifie les conditions nécessaires pour que la porte génère une transition descendante.

Soit un ensemble de signaux S.

Définition 3.6 *Une règle de production sur un signal $s \in S$ est constituée de deux éléments :*

- *une garde, qui est une expression booléenne des éléments de $S - s$,*
- *une direction de transition, notée $\nearrow$ ou $\searrow$, qui spécifie si la transition gardée est montante ou descendante.*

Elle est représentée de la façon suivante, $f(a, b, \ldots)$ étant la garde (fonction booléenne des signaux $a, b, \ldots$) et s le signal modélisé :

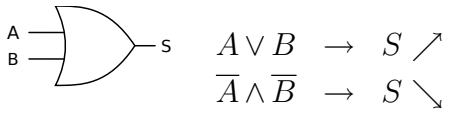
$$f(a, b, \ldots) \quad \rightarrow \quad s \nearrow$$

La garde spécifie les conditions que le circuit doit respecter pour que la transition considérée puisse avoir lieu. On n'autorise pas le comportement de la porte modélisée à dépendre de la sortie de cette porte (sortie rebouclée sur une entrée). En effet, une telle boucle permettrait d'avoir une porte à mémoire, mais les règles de productions permettent déjà de modéliser de telles portes, sans avoir à utiliser un tel artifice.

Il faut un couple de règles de productions pour modéliser une porte à une sortie. L'une des règles de transition modélise les transitions montantes de la sortie, et l'autre modélise les transitions descendantes. Plus généralement, il faut $2n$ règles de productions pour modéliser une porte à n sorties.

La figure 3.9 montre comment sont modélisées les portes OR et Muller avec des règles de productions. On peut remarquer que dans le cas de portes sans mémoire, comme la porte OR, on peut passer de la garde de la transition montante à la garde de la transition descendante par négation logique. En effet, comme la porte est sans mémoire, l'état de la sortie correspond à la valeur de la fonction booléenne calculée, ici un *ou logique*. Ainsi, la porte ne peut émettre une transition montante que si la fonction booléenne est vraie. De même, la porte ne peut émettre une transition descendante que si la fonction booléenne est fausse, c'est-à-dire si sa négation logique est vraie.

Définition 3.7 *Un couple de règles de production* $B^+ \rightarrow x \nearrow, B^- \rightarrow x \searrow$ *est* ***combinatoire*** *si et seulement si* $B^+ = \overline{B^-}$.



(a) porte OR (b) porte de Muller

FIG. 3.9 – Exemples de modélisation de portes par des règles de productions

4 Conclusion

Dans ce chapitre, on a présenté les différents modèles de circuits asynchrones nécessaires pour la suite. Il existe deux catégories de modèles de circuit : les modèles comportementaux, qui spécifient quel comportement le circuit doit adopter, en fonction de ses entrées, et les modèles structurels, qui spécifient la structure du circuit, c'est-à-dire les composants qui constituent le circuit.

Le premier modèle comportemental présenté est le modèle CHP : c'est le langage de spécification utilisé dans TAST. Basé sur le langage CSP, il modélise des processus concurrents, qui communiquent via des canaux de communication. Un programme CHP n'est pas toujours synthétisable, à cause de la possibilité de créer des séquencialités artificiellement dans le programme. Il est possible de transformer un programme CHP non-synthétisable en un programme synthétisable, en décomposant les processus qui posent problème. Un programme qui est ainsi raffiné est dit DTL.

Le modèle de MDD, comportemental lui aussi, a été développé au cours de cette thèse. Il est le cœur de la méthode de synthèse. Il est basé sur les diagrammes de décisions multi-valués, auxquels on a ajouté une sémantique permettant de décrire le circuit : en particulier, il ne spécifie pas seulement la fonction logique à implémenter, mais aussi la consommation ou non des variables d'entrée.

Le modèle structurel que l'on utilise par la suite, la netlist, est défini formellement. Ce modèle décrit un réseau de portes. Il peut être rendu hiérarchique, ce qui est nécessaire pour traiter de gros circuits. Il permet de décrire et manipuler le circuit issu de la synthèse.

Un autre modèle structurel est utilisé : le modèle de règles de production. Ce modèle, permet de décrire formellement les portes logiques du circuit, et leur interconnexion. Il est nécessaire pour l'étude théorique du chapitre 5.

Chapitre 4

Synthèse et optimisation de circuits QDI

1 Introduction

Le chapitre précédent présente, parmi les différents modèles de circuits QDI, le modèle de MDD, modèle autour duquel s'articule la méthode de synthèse développée dans cette thèse. L'intérêt et l'innovation de cette méthode réside dans le fait que les circuits synthétisés sont garantis QDI, sans avoir à faire de vérification de cette propriété dans le flot de synthèse.

Toutes les étapes de transformation du circuit sont effectuées sur les MDDs. Il est prouvé formellement que les circuits générés à partir de notre modèle à base de MDD, avant ou après transformation, sont QDI.

Ce chapitre présente les différentes manipulations que l'on effectue sur les MDDs. Tout d'abord, le MDD est généré à partir d'une description comportementale du circuit, sous forme d'un programme en langage CHP. Le MDD est ensuite transformé pour être optimisé. Le but de cette étape est d'éliminer les redondances en factorisant les parties communes, et ainsi réduire la taille du circuit. Pour cela, plusieurs algorithmes d'optimisation sont appliqués.

Enfin, le MDD est synthétisé, c'est-à-dire qu'un réseau de portes est généré. Deux algorithmes de synthèse de MDD sont présentés : tout d'abord l'algorithme DIMS, bien connu, qui est très simple à mettre en œuvre mais génère une netlist peu utilisable, puis l'algorithme de synthèse en portes à deux entrées, qui fait une décomposition automatique des portes. Cette décomposition est essentielle pour pouvoir effectuer une projection technologique du circuit.

2 Génération du MDD

Dans le flot de synthèse de l'outil TAST, le MDD est généré à partir du programme CHP décrivant le circuit. L'automatisation de cette étape, qui est un problème complexe, est en cours de finalisation. Cette section présente l'approche retenue en l'état des choses.

Le principal problème vient du fait que le chemin d'acquittement est difficile à extraire du programme CHP : il est spécifié de manière indirecte, via les actions de lecture et d'écriture sur les canaux de communication, et incomplète : seule la

phase montante du protocole est spécifiée. Le CHP ne spécifie pas à quand il faut acquitter les entrées lues, ni quand il faut remettre à zéro les sorties émises, tant que les canaux respectent le protocole quatre phases, et que toutes les entrées lues sont acquittées avant le prochain cycle de calcul.

La solution retenue consiste à passer par une représentation flot de donnée du programme CHP, en ignorant les «fausses» dépendances créées par les opérateurs séquentiels : seules les dépendances de données sont considérées. C'est à partir du flot de données qu'est calculé le chemin d'acquittement du circuit : ce chemin est en quelque sorte le flot opposé : les acquittements remontent le flot de données, des sorties vers les entrées. Un ensemble de règles locales permettent de construire systématiquement ce flot d'acquittement à partir du flot direct.

À partir du flot de données, un ensemble de règles permet de calculer l'équation logique de chaque sortie et chaque acquittement d'entrée. En fait, l'équation générée comporte plus d'information qu'une simple équation booléenne, puisque comme on l'a vu au chapitre précédent, il est nécessaire de garder l'information de consommation des variables pour construire un MDD. À partir de ces équations, il est relativement facile de générer un MDD.

3 Synthèse basique du chemin de données : DIMS

3.1 Présentation de la synthèse DIMS

L'algorithme DIMS (Delay Insensitive Minterms Synthesis, c'est-à-dire Synthèse Insensible aux Délais de Minterms) effectue une synthèse de circuits QDI de la manière la plus simple qui soit : cet algorithme implémente directement la table de vérité de la fonction booléenne que l'on souhaite réaliser [37, 38].

Pour chaque ligne de cette table de vérité, c'est-à-dire pour chaque minterme de l'équation booléenne, une porte de Muller est créée, qui regroupe le rail considéré de chaque entrée. Ainsi, le minterme $A_0B_1C_0$ correspond à une porte de Muller à 3 entrées, qui regroupe le rail 0 de l'entrée A, le rail 1 de l'entrée B et le rail 0 de l'entrée C.

Puis, pour chaque valeur de la sortie (c'est-à-dire chaque rail de la sortie, puisque celle-ci est codée en 1-parmi-n), une porte OR est créée, qui regroupe les mintermes pour lesquels la fonction prend la valeur considérée.

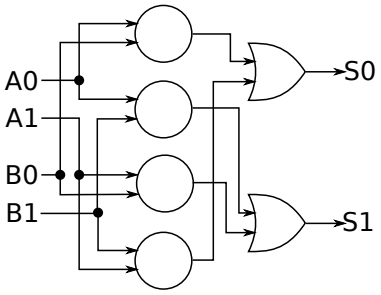
La figure 4.1 montre un exemple de circuit issu d'une synthèse DIMS. L'équation booléenne est $S = A \operatorname{xor} B$, c'est-à-dire, en décomposant en mintermes, $S = \overline{A} \wedge B \vee A \wedge \overline{B}$.

L'algorithme de synthèse DIMS génère des circuits à deux couches : une couche de portes de Muller et une couche de portes OR. Ces circuits sont particulièrement rapides, puisque le chemin des données entre les entrées et les sorties ne traverse que deux portes logiques. Cependant, la taille du circuit croît exponentiellement avec le nombre d'entrées : un circuit DIMS à i digits d'entrée, codées en 1-parmi-n, comporte n^i portes de Muller.

Un tel algorithme de synthèse n'est donc utilisable que pour les circuits comportant très peu d'entrées (deux, éventuellement trois). Si le processus comporte un plus grand nombre d'entrées, on souhaite avoir à notre disposition un algorithme de synthèse plus efficace, qui génère un circuit d'une taille raisonnable. Pour cela, il faut

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

(a) table de vérité



(b) circuit

FIG. 4.1 – Exemple de synthèse DIMS : $S = A \operatorname{xor} B$. Le circuit est directement synthétisé à partir de la table de vérité.

factoriser les parties communes, afin de générer des circuits à plusieurs niveaux [39]. Cependant, ceci pose des problèmes du point de vue de l'insensibilité aux délais, comme le montre la section 5.1, plus bas.

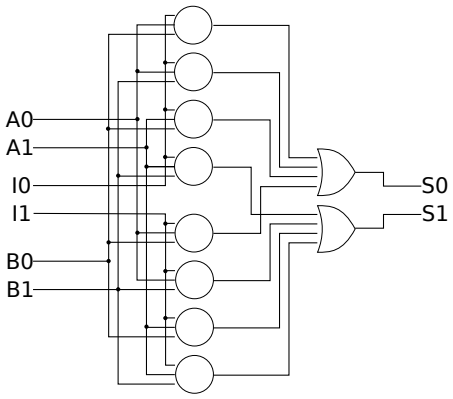


FIG. 4.2 – Le nombre de portes de Muller dans un circuit généré par l'algorithme de synthèse DIMS est une fonction exponentielle du nombre d'entrées : pour 3 entrées booléennes, il y a $2^3 = 8$ portes de Muller.

3.2 Algorithme de synthèse DIMS à partir d'un MDD

L'algorithme de synthèse DIMS est très simple : il consiste à parcourir l'ensemble des chemins du MDD (ce qui se fait facilement par un parcours en profondeur de l'arbre à partir de chaque racine : chaque feuille que l'on atteint correspond à un chemin du graphe. Les fonctions *var* et *val*, définies au chapitre 3, renvoient respectivement la variable et la valeur d'un tronçon de chemin, c'est-à-dire la variable étiquetant le nœud du tronçon, et le numéro du fils de ce terminal qu'il faut suivre pour continuer sur le chemin.

Algorithme 3.1

```
fonction synthèse_DIMS(mdd) :
    // initialisation
    pour chaque rail i de chaque sortie s :
        s[i] = liste_vide
```

```
// création des portes de Muller
pour chaque chemin c du mdd : // parcourt en profondeur du mdd
    n = taille(c) // longueur du chemin c, en nombre de nœuds
    m = créer une porte de Muller à n entrées
    pour chaque tronçon t_i de c :
        relier l'entrée i de m au rail val(t_i) de la variable var(t_i)
    f = feuille(c) // feuille terminale du chemin
    ajouter à la liste var(f)[val(f)] la sortie de m
// Regroupement des portes de Muller par sortie, à l'aide de portes OR
pour chaque rail i de chaque sortie s :
    n = taille(s[i]) // nombre d'éléments dans la liste s[i]
    si n=1 :
        relier l'unique fil dans la liste s[i] au rail i de la sortie s.
    sinon:
        o = créer une porte OR à n entrées
        relier la sortie de o au rail i de la sortie s
        pour chaque w dans la liste s[i] :
            relier le fil w à une entrée o
```

On note que le circuit généré par cet algorithme respecte la sémantique du MDD spécifiée dans le chapitre précédent. En effet, puisqu'une porte de Muller est générée pour chaque chemin du circuit, rien ne se passe tant que l'ensemble des nœuds d'un chemin n'est pas activé, et tous les fils sont désactivés (à 0). Lorsque tous les nœuds d'un chemin sont activés, la porte de Muller commute, la porte OR connectée aussi puisqu'elle était à 0.

De même, lorsqu'un chemin a été activé, rien ne se passe tant que tous les nœuds de ce chemin ne sont pas désactivés. Lorsqu'ils sont tous désactivés, la porte de Muller commute pour repasser à 0, et de même la porte OR repasse à 0 (puisque les chemins sont mutuellement exclusifs, une seule porte de Muller pouvait être à 1, donc si celle-ci repasse à 0, toutes les entrées de la porte OR sont à 0).

4 Optimisations du MDD

4.1 Nécessité de l'optimisation

Le principal inconvénient des circuits QDI est leur taille : ils contiennent beaucoup plus de transistors que les circuits équivalents synchrones. Ceci est du en partie au fait que l'on utilise le codage 1-parmi-n : là où un circuit synchrone utilise un fil, et ne calcule qu'une fonction logique (il ne calcule que la valeur 1 de la sortie), le circuit QDI codé en 1-parmi-2 utilise deux fils, et calcule deux fonctions logiques (une pour chaque valeur possible de la sortie). De plus, le circuit QDI intègre un calcul de l'acquittement, qui n'existe pas en synchrone.

Cependant, ces explications ne sont pas suffisantes pour expliquer la différence de taille. En fait, les circuits QDI développés actuellement sont peu optimisés ; il y a de nombreuses optimisations possibles qui ne sont pas faites, contrairement aux circuits synchrones, pour lesquels de nombreuses techniques d'optimisation approchant le circuit optimal ont été développées. Il est donc nécessaire de développer

des techniques d'optimisation efficaces si l'on veut rendre la technologie asynchrone quasi insensible aux délais attrayante pour l'industrie.

Cette section présente les techniques d'optimisation développées dans cette thèse. Comme cela a été expliqué, toute l'optimisation est faite sur le modèle de MDD, afin de garantir que le circuit synthétisé est QDI.

Les techniques mises en œuvre dans cette thèse ne sont pas optimales, et certaines optimisations restent possibles. Elles permettent néanmoins déjà de faire des circuits plus petits qu'avec l'ancienne version de TAST. De plus, la méthode employée pour développer ces techniques est réutilisable : elle consiste à développer un algorithme de transformation des MDDs, qui réduit la taille du MDD, et à prouver que les MDDs transformés sont sémantiquement équivalents aux MDDs avant transformation.

4.2 Algorithmes d'optimisation

Hypothèses d'optimisation

Les algorithmes présentés ici transforment le MDD, et non directement le circuit. Ceci est à la fois nécessaire pour garantir que le circuit généré après optimisation est QDI, et problématique pour évaluer l'optimisation : le circuit généré à partir du MDD transformé est-il plus petit ou plus gros que le circuit généré à partir du MDD initial ? Rien ne permet de le dire, à moins de générer explicitement les deux.

Afin de résoudre ce problème sans explicitement générer le circuit à chaque étape, nous faisons l'hypothèse suivante : la taille du circuit généré est proportionnelle au nombre de nœuds du MDD. Cette hypothèse, simplificatrice, permet d'évaluer immédiatement l'efficacité d'une étape d'optimisation.

Cette hypothèse est valide, contrairement au synchrone, où rien ne permettrait d'affirmer que la taille du circuit est liée à celle du BDD le modélisant. En effet, ici, la synthèse en portes deux entrées génère des circuits dont la structure est proche du MDD, il est donc naturel que leurs tailles soient liées. De plus, on s'interdit toute optimisation logique sur le circuit : la structure du circuit ne sera donc jamais modifiée, et restera liée à celle du MDD.

Réduction

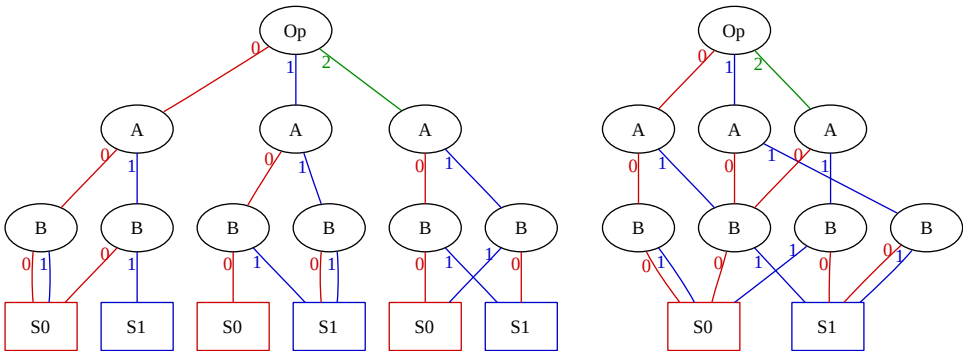
L'algorithme de réduction, permet de fusionner les nœuds du MDD qui sont fonctionnellement équivalents, c'est-à-dire les nœuds qui sont construits de manière identique à partir des terminaux (c'est-à-dire que les sous-graphes contenant les chemins entre le nœud en question et les terminaux sont identiques).

Définition 4.1 *Deux nœuds a et b sont* ***fonctionnellement équivalents*** *si et seulement si l'une des conditions ci-dessous est vérifiée :*

- *a et b sont des terminaux, $var(a) = var(b)$ et $val(a) = val(b)$,*
- *a et b sont des non-terminaux, $var(a) = var(b)$ et $\forall i \in range(var(a)), child(a, i)$ est fonctionnellement équivalent à $child(b, i)$,*
- *a et b sont des fourches, et $\forall i \in range(var(a)), child(a, i)$ est fonctionnellement équivalent à $child(b, i)$.*

Cette définition est récursive ; elle termine toujours puisque le MDD est un graphe acyclique.

La figure 4.3 illustre cette transformation sur un exemple.



(a) MDD avant réduction (b) MDD après réduction

FIG. 4.3 – exemple de réduction de MDD

L'algorithme parcourt le mdd en profondeur d'abord, en mémorisant les nœuds déjà parcourus dans une table de hachage. En fait, 3 tables de hachages distinctes sont utilisées : une pour les terminaux, une pour les non-terminaux et une pour les fourches. La clé utilisée pour insérer un nœud n à la table de hachage comporte les valeurs permettant de repérer les nœuds fonctionnellement équivalents : pour un terminal $var(n)$ et $val(n)$, pour un non-terminal, $var(n)$ ainsi que la référence de chacun des fils, et pour une fourche, la référence de chacun des fils. Ainsi, deux nœuds fonctionnellement équivalents accèdent à la même case de la table de hachage, dans laquelle on stocke la référence du nœud que l'on souhaite garder : s'il existe déjà un nœud fonctionnellement équivalent au nœud parcouru, on remplace le nœud parcouru par ce nœud ; sinon, on ajoute le nœud parcouru dans la table correspondante, et on continue le parcours.

Algorithme 4.1

```
fonction réduction(mdd) :
    terminaux = table de hachage de nœuds
    nonterminaux = table de hachage de nœuds
    fourches = table de hachage de nœuds
    pour chaque racine n de mdd :
        réduction_nœud(n, terminaux, nonterminaux, fourches)
fonction réduction_nœud(n, terminaux, nonterminaux, fourches) :
    // le MDD est un graphe et non un arbre, on ne veut parcourir qu'une fois
    chaque nœud.
    si done(n) = 1 :
        retourner
    sinon:
        done(n) = 1
    si n est un terminal :
        si n' = terminaux[var(n), val(n)] existe :
            remplacer n par n'
        sinon:
            terminaux[var(n), val(n)] = n
```

```
si n est un non-terminal :
    si n' = nonterminaux[var(n), childs(n)] existe :
        remplacer n par n'
    sinon:
        nonterminaux[var(n), childs(n)] = n
        // appel récursif, pour continuer le parcours en profondeur
        pour i ∈ range(var(n)) :
            réduction_nœud(child(i, n), terminaux, nonterminaux, fourches)
si n est une fourche :
    si n' = fourches[childs(n)] existe :
        remplacer n par n'
    sinon:
        fourches[childs(n)] = n
        // appel récursif, pour continuer le parcours en profondeur
        pour c ∈ childs(n) :
            réduction_nœud(c, terminaux, nonterminaux, fourches)
```

Cet algorithme substitue des nœuds du MDD qui sont fonctionnellement équivalents, la sémantique du MDD n'est donc pas modifiée par cette transformation. Si l'on suppose que les accès aux tables de hachage se font en temps constant, la complexité de cet algorithme est $\mathcal{O}(n + v)$, où n est le nombre de nœuds du MDD et v son nombre d'arcs. En effet, l'algorithme fait un parcours en profondeur du MDD, donc il parcourt chaque nœud, et chaque arc sortant de chaque nœud.

Concernant le nombre de nœuds du circuit, il décroît forcément, puisqu'aucun nouveau nœud n'est créé, et que certains nœuds sont supprimés.

Cet algorithme est donc rapide, peu coûteux, et optimise toujours le MDD.

Factorisation

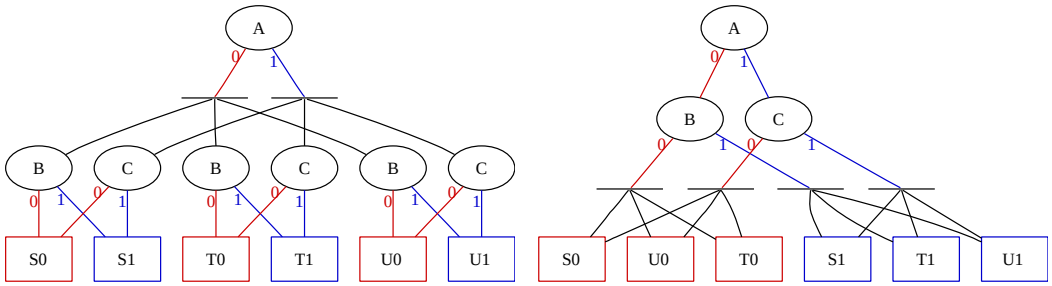
Contrairement à l'algorithme de réduction, l'algorithme de factorisation est spécifique aux MDD. Plus précisément, il est lié à l'existence de fourches dans les MDDs : son but est de fusionner les nœuds similaires issus d'une même fourche. Deux nœuds sont similaires s'ils sont du même type et ont les mêmes étiquettes :

Définition 4.2 *Deux nœuds a et b sont* ***similaires*** *si et seulement si l'une des conditions suivantes est vraie :*

- *a et b sont des terminaux, $var(a) = var(b)$ et $val(a) = val(b)$,*
- *a et b sont des non-terminaux, et $var(a) = var(b)$,*
- *a et b sont des fourches.*

L'algorithme parcourt en profondeur le MDD. À chaque fourche, il analyse les nœuds fils : il fusionne les nœuds similaires, comme le montre la figure 4.4 qui illustre cette transformation sur un exemple.

La détection de nœuds similaires utilise une table de hachage ; la clé utilisée dans la table utilise la variable du non-terminal. Les fourches sont toutes similaires, il n'y a donc pas besoin de table de hachage pour les fusionner. Les terminaux similaires sont déjà fusionnés par l'algorithme de réduction, étant donné que deux terminaux fonctionnellement équivalents sont similaires.



(a) MDD avant factorisation (b) MDD après factorisation

FIG. 4.4 – exemple de factorisation de MDD

Algorithme 4.2

```
fonction factorisation(mdd) :
    pour chaque racine n de mdd :
        factorisation_nœud(n)
fonction factorisation_nœud(n) :
    // le MDD est un graphe et non un arbre, on ne veut parcourir qu'une fois
    chaque nœud.
    si done(n) = 1 :
        retourner
    sinon:
        done(n) = 1
    si n est un non-terminal :
        // parcours en profondeur
        pour i ∈ range(var(n)) :
            factorisation_nœud(child(i, n))
    si n est une fourche :
        p = partitionner_nœuds_similaires(childs(n))
        pour l ∈ p :
            si il y a plus d'un nœud dans l :
                supprimer les nœuds de l dans childs(n)
                a = fusionner(l)
                ajouter a dans childs(n)
        // parcours en profondeur :
        pour c ∈ childs(n) :
            factorisation_nœud(c)
fonction partitionner_nœuds_similaires(noeuds) :
    // renvoie une liste de listes de nœuds identiques
    h = table de hachage de listes de nœuds
    fourches = liste_vide
    terminaux = liste_vide
    pour chaque n ∈ noeuds :
        si n est un non-terminal :
            ajouter n à la liste h[var(n)]
        si n est une fourche :
            ajouter n à la liste fourches
```

```
            si n est un terminal :
                ajouter n à la liste terminaux
        a = liste_vide
        pour chaque l ∈ h :
            ajouter l à a
        ajouter fourches à a
        pour chaque n ∈ terminaux :
            ajouter n à a
        retourner a
    fonction fusionner(l) :
        si les éléments de l (tous similaires) sont des non-terminaux :
            a = nouveau non terminal similaire aux éléments de l (même variable)
            pour i ∈ range(var(a)) :
                child(i, a) = nouvelle_fourche()
                pour n ∈ l :
                    ajouter child(i, n) à childs(child(i, a))
            retourner a
        si les éléments de l (tous similaires) sont des fourches :
            f = nouvelle_fourche()
            pour n ∈ l :
                ajouter les éléments de childs(n) à childs(f)
            retourner f
        si les éléments de l (tous similaires) sont des terminaux :
            a = nouveau terminal similaire aux éléments de l (même variable et
            valeur)
            retourner a
```

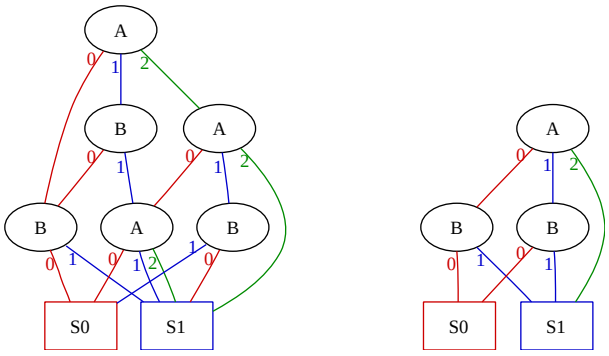
On vérifie que la transformation de factorisation préserve la sémantique du MDD. Soient a et b deux nœuds similaires issus d'une même fourche f : $var(a) = var(b)$. la condition pour que a soit actif est identique à la condition pour que b soit actif : il faut que f soit active, et que la variable $var(a)$ soit valide. a et b sont donc toujours actifs en même temps. De plus, comme a et b sont similaires, leur comportement est le même, et donc pour $i \in range(var(a))$, $child(i, a)$ et $child(i, b)$ sont toujours actifs en même temps : tout se passe comme si $child(i, a)$ et $child(i, b)$ étaient issus d'une même fourche.

Ainsi, si l'on fusionne a et b en un nouveau nœud c tel que $var(c) = var(a) = var(b)$, et que, pour $i \in range(var(c))$, on regroupe $child(i, a)$ et $child(i, b)$ par une fourche $f_i = child(i, c)$, alors le comportement du MDD ne change pas : la sémantique du MDD est préservée.

Les fonctions *partitionner_nœuds_similaires* et *fusionner* sont appelées chacune une fois par fourche du MDD, donc moins de $\mathcal{O}(n)$ fois, n étant le nombre de nœuds du MDD. Ils bouclent sur les fils de la fourche considérée, donc moins de $\mathcal{O}(n)$ fois, ce qui donne une complexité de l'algorithme de $\mathcal{O}(n^2)$. Cependant, ce pire cas est très rarement atteint, et très loin du cas moyen : si l'on instaure une limite B telle que pour chaque nœud n, il y a moins de B fourches ayant pour fils n, alors chaque nœud ne sera parcouru en tout que B fois au maximum. Ainsi, la complexité est $\mathcal{O}(B \times n)$. En général, B est très faible, cet algorithme est donc quasiment linéaire par rapport à la taille du MDD.

Unicité

L'algorithme d'unicité permet de s'assurer que sur chaque chemin du MDD, chaque variable est unique. D'après la sémantique que l'on a définie pour un MDD, il n'y a pas d'intérêt à avoir une même variable étiquetant plusieurs nœuds sur un même chemin du MDD. En effet, pendant un cycle de calcul, une variable, si elle est valide, a une unique valeur. Ainsi, soit les deux nœuds non-terminaux étiquetés par cette variable ont la même valeur sur le chemin considéré, et ils sont redondants puis actifs tous les deux en même temps, soit ils ont des valeurs distinctes, et le chemin ne pourra jamais être actif : il est alors inutile.



(a) MDD avant unicité

(b) MDD après unicité

FIG. 4.5 – exemple d'unicité de MDD

L'algorithme d'unicité consiste en un parcours des chemins du MDD, en mémorisant les variables traversées sur le chemin courant, ainsi que les valeurs associées, dans une pile : à chaque fois que l'on descend dans l'arbre via un non-terminal (on allonge le chemin courant), on empile un nouveau couple constitué de la variable et sa valeur, et à chaque fois que l'on remonte sur un non-terminal, (on raccourcit le chemin courant), on dépile. Pour chaque non-terminal parcouru, on vérifie qu'il n'est pas déjà présent dans notre pile. Si c'est le cas, le nœud est redondant. La valeur associée à la variable dans la pile indique le seul fils intéressant de ce nœud : les autres chemins issus de ce nœud sont inutiles.

Algorithme 4.3

```
fonction unicité(mdd) :
    pile = pile_vide
    pour chaque racine n du mdd :
        unicité_nœud(n)
fonction unicité_nœud(n, pile) :
    si n est un non-terminal :
        si var(n) est dans pile avec la valeur i :
            remplacer n par child(i, n) dans le mdd
        sinon:
            pour chaque i ∈ range(var(n)) :
                empiler(var(n), i) dans pile
                unicité_nœud(child(i, n), pile)
                dépiler(pile)
```

```
si n est une fourche :
    pour chaque c ∈ childs(n) :
        unicité_nœud(n, pile)
```

Cette transformation préserve la sémantique du MDD, puisque seuls les chemins qui ne peuvent jamais devenir actifs sont supprimés.

L'algorithme fait un parcours des chemins du graphe, il peut donc parcourir plusieurs fois chaque nœud. Au pire cas, on a moins de $d \times c$ opérations, où d est la profondeur maximale du graphe (généralement $d = \log(n)$), et c le nombre de chemins du MDD. À chaque itération, on effectue une recherche (linéaire) dans la pile, qui contient au plus d éléments. La complexité de l'algorithme est donc $\mathcal{O}(d^2 \times c)$. c peut exploser : dans le pire cas, c est une fonction exponentielle du nombre de nœuds. Ce pire cas ne se produit que rarement, mais l'algorithme peut être coûteux.

5 Limitation du fan-in : synthèse en portes à deux entrées

5.1 Intérêt de la limitation du fan-in

La section 3 a présenté l'algorithme de synthèse DIMS, qui génère un circuit (c'est-à-dire une netlist de portes) à partir d'un MDD. Cependant, cette netlist a de nombreux défauts. Tout d'abord, elle est très grosse : le nombre de portes générées est proportionnel au nombre de chemins du MDD, et explose donc avec le nombre de nœuds du MDD. De plus, les portes ont un fan-in non borné. Le fan-in d'une porte est le nombre d'entrées que cette porte contient. Dans une netlist générée par l'algorithme DIMS, les portes de Muller ont autant d'entrées que la taille du chemin correspondant.

Cependant, il n'est pas possible en pratique d'utiliser des portes de fan-in aussi grand que l'on veut. En effet, la netlist se base sur une bibliothèque de cellules : chaque porte de la netlist est l'instance d'une cellule de la bibliothèque. La bibliothèque contient un nombre fini de cellules, et se limite aux cellules avec un petit fan-in (jusqu'à 4 ou 5).

Gestion habituelle du problème de fan-in : projection technologique

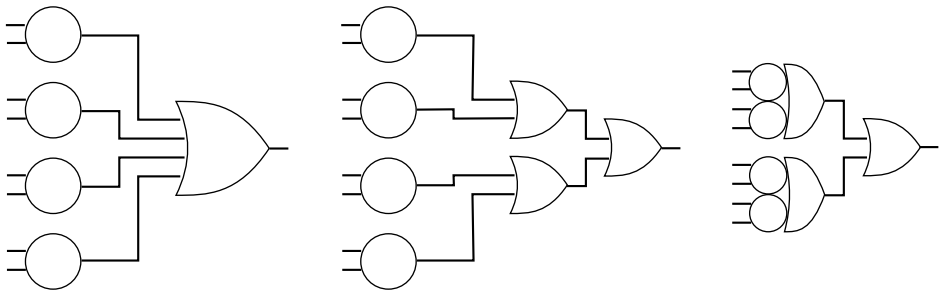
La méthode habituelle pour régler ce problème consiste à séparer la synthèse en deux parties :

- La première étape fait la synthèse du circuit sans contrainte de fan-in.
  Pour cela, on utilise une bibliothèque «virtuelle», contenant les portes de base pour toutes les valeurs possibles du fan-in. Cette bibliothèque est virtuelle puisqu'elle continuent un nombre infini de portes (il n'y a pas de limite au fan-in). Chaque porte est décrite uniquement de manière fonctionnelle, et donc indépendemment du nombre d'entrées.
- La seconde étape décompose les portes virtuelles afin de n'utiliser que des cellules de la bibliothèque cible, puis elle les regroupe afin de réduire le nombre de cellules utilisées, lorsque c'est possible. On parle de projection technologique [40, 41].

Cependant, cette méthode n'est pas applicable dans notre flot de synthèse. En effet, on veut être certain qu'après synthèse, les manipulations faites sur la netlist de portes ne compromettent pas la quasi insensibilité aux délais du circuit. La netlist issue de la synthèse est garantie QDI ; mais si cette netlist est transformée sans faire attention, rien ne garantit qu'elle reste QDI.

En fait, il y a deux types de transformations que l'on souhaite effectuer sur la netlist pour effectuer la projection technologique, comme l'illustre la figure 4.6 :

- décomposer une porte, c'est-à-dire la remplacer par deux portes ou plus fonctionnellement identiques
- et fusionner deux portes ou plus, c'est-à-dire les remplacer par une porte fonctionnellement identique.



(a) avant projection technologique (b) décomposition (c) fusion

FIG. 4.6 – La projection technologique décompose les portes logiques : la porte OR à quatre entrées de la figure (a) est décomposée en 3 portes OR à deux entrées, figure (b). Puis elle fusionne les portes : deux portes de Muller et une porte OR sont fusionnées en une seule porte complexe dans la figure (c).

La fusion ne pose pas de problèmes, puisque l'on supprime des nets, sans changer fonctionnellement le circuit : les contraintes sur les nets restants sont les mêmes, le circuit est donc toujours QDI. On peut noter qu'en fait, on ne supprime pas réellement le net : on l'inclut juste dans la porte complexe de la bibliothèque ciblée. Mais les nets à l'intérieur des cellules de la bibliothèque ne sont pas le problème de l'outil de synthèse : c'est au concepteur de la bibliothèque de se soucier que sa cellule fonctionne correctement, quels que soient les délais. Du point de vue de l'outil de synthèse, les cellules de la bibliothèque sont le niveau d'abstraction le plus bas que l'on considère ; de ce point de vue, le net est donc effectivement supprimé.

Par contre, la décomposition peut poser des problèmes [42]. Si la porte que l'on décompose est une porte purement combinatoire (AND, OR, ou combinaison), l'ensemble de portes décomposées est strictement équivalent à la porte initiale, et tout comme dans le cas de la fusion, il n'y a pas de problèmes. Mais dans le cas de décomposition de portes de Muller, il n'y a pas équivalence, et le circuit peut perdre sa quasi insensibilité aux délais.

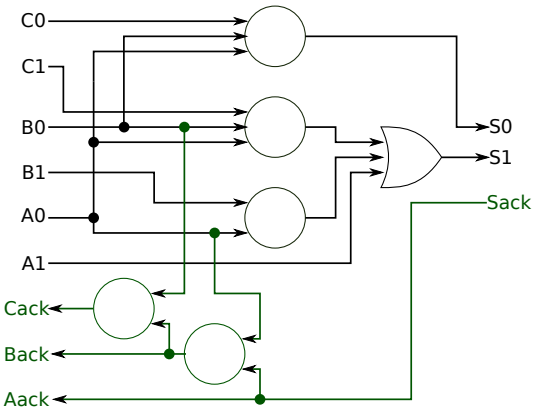
Problème de décomposition des portes de Muller

La figure 4.7(b) montre un exemple de circuit QDI pour lequel on ne peut pas décomposer les portes de Muller à 3 entrées : le résultat, présenté figure 4.7(c), n'est pas QDI. En effet, si le circuit décomposé reçoit A_0 et B_0, les deux portes de Muller correspondantes vont commuter, mais seule l'une d'entre elles participera à la transition de la sortie (selon la valeur lue sur C). Ainsi, l'une des transitions en sortie des portes de Muller ne sera pas acquittée, et risque de ne pas être remise à zéro lors de la phase de remise à zéro du circuit, ce qui compromet le bon fonctionnement ultérieur du circuit.

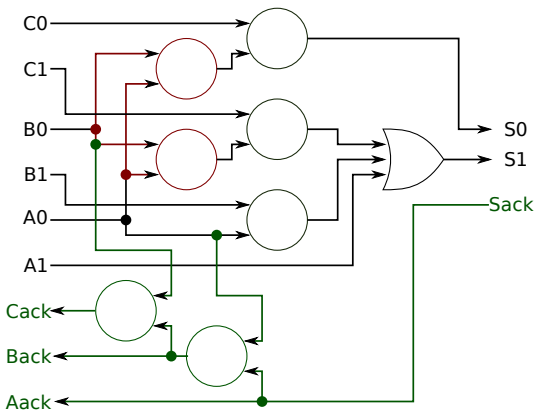
Pour résoudre ce problème, il faut fusionner les deux portes de Muller crées ayant les mêmes entrées, comme le présente la figure 4.7(d). Lorsque ce circuit reçoit A_0 et B_0, une seule porte de Muller commute. La fourche en sortie de cette porte est isochrone ; une seule de ses branches participe à la transition de la sortie, ce qui est autorisé dans la classe des circuits quasi insensibles aux délais. La transition en sortie de la porte de Muller est donc bien acquittée, elle sera remise à zéro et le circuit fonctionne correctement.

```
A ?a ;
@[ a=0 => B ?b ;
          @[ b=0 => C ?c ; S !c ; break
             b=1 => S !1 ; break ]
   a=1 => S !1 ; break ] ;
loop
```

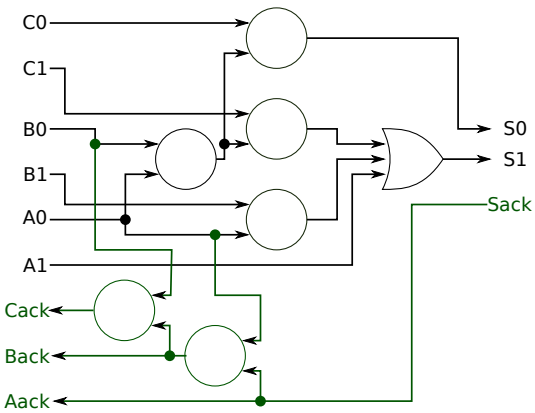
(a) code CHP



(b) circuit non décomposé (DIMS)



(c) circuit mal décomposé ; non QDI



(d) circuit décomposé correct

FIG. 4.7 – La décomposition de portes de Muller peut donner un circuit qui n'est plus QDI.

Solution adoptée : synthèse en portes à deux entrées

Le problème ci-dessus montre qu'on ne peut pas autoriser l'algorithme de projection technologique à décomposer les portes : celui-ci, qui agit localement sur un petit ensemble de portes à la fois, sans avoir de vision globale du circuit, ne pourrait pas garantir qu'il ne compromet pas la quasi insensibilité aux délais du circuit en faisant une décomposition.

Afin de résoudre ce problème, l'approche envisagée dans notre flot de synthèse est légèrement différente de celle présentée dans la section 5.1 : l'étape de décomposition des portes est intégrée à la synthèse, qui est la seule étape capable de faire cette décomposition tout en garantissant la quasi insensibilité aux délais du circuit. En fait, le circuit est directement synthétisé sous sa forme décomposée. L'algorithme de projection technologique effectue uniquement des fusions de portes.

Cependant, au niveau de la synthèse, il n'est pas possible de connaître les portes qui nécessiteront une décomposition : c'est la projection technologique qui fait ces choix. Pourtant, on peut noter qu'une décomposition inutile peut être annulée par une fusion : il n'est pas dangereux de décomposer plus de portes que nécessaire, l'algorithme de projection technologique pourra fusionner les portes qui avaient été inutilement décomposées.

La solution consiste alors à décomposer au maximum chaque porte du circuit lors de la synthèse : le circuit issu de la synthèse ne comporte que des portes à deux entrées. Ainsi, en faisant uniquement des fusions, l'algorithme de projection technologique peut optimiser le circuit sans jamais risquer de compromettre sa quasi insensibilité aux délais.

5.2 Algorithme de synthèse de MDD en portes à deux entrées

L'algorithme de synthèse de MDD en portes à deux entrées se base sur la structure de graphe du MDD, qu'il préserve : chaque nœud du graphe est synthétisé en un ensemble d'instances, et chaque arc du MDD est synthétisé en un net. La figure 4.8 montre comment les nœuds du MDD sont transformés.

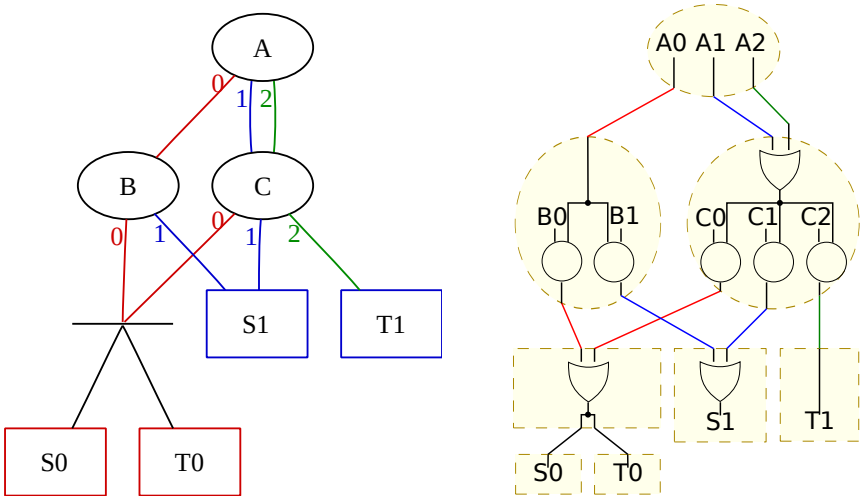
Soit n un nœud du MDD.

- Les arcs entrants d'un nœud sont regroupés par un arbre de portes OR, pour obtenir un rail, appelé activation du nœud.
- Un terminal correspond directement à un rail d'un port primaire de la netlist : l'activation de n est directement reliée au rail $val(n)$ du canal $var(n)$.
- Une fourche correspond à une fourche dans la netlist, c'est-à-dire à un ensemble de nets ayant le même port de départ, c'est-à-dire sortant de la même instance : chaque arc sortant de n est relié à l'activation de n.
- Un non-terminal est implémenté par une structure plus complexe, qui sépare l'activation de n et la combine à chaque rail de la variable $var(n)$ avec une porte de Muller. La sortie de chaque porte de Muller correspond à l'un des arcs sortants de n.

Bien entendu, un non-terminal racine n'a pas d'activation ; il n'y a donc pas de portes de Muller à créer, chaque rail de la variable $var(n)$ correspond directement à l'un des arcs sortants.

L'algorithme est le suivant :

Algorithme 5.1



(a) le MDD (b) la netlist synthétisée

FIG. 4.8 – La synthèse d'un MDD en portes à deux entrées préserve la structure du MDD : chaque nœud est synthétisé en un ensemble d'instances, et chaque arc en un net.

```
fonction synthèse(mdd) :
    // initialisation
    pour chaque nœud n du mdd :
        activation_in[n] = liste_vide
        activation[n] = nouveau_net()
    // création des portes de Muller
    pour chaque nœud n du mdd :
        si n est un terminal :
            relier activation[n] au rail val(n) de var(n)
        si n est un non-terminal :
            pour pour i ∈ range(var(n)) :
                m = créer une porte de Muller à 2 entrées
                relier une entrée de m au rail i de var(n)
                relier l'autre entrée de m à activation[n]
                ajouter à la liste activation_in[child(i, n)] la sortie de m
        si n est une fourche :
            pour chaque pour chaque c ∈ childs(n) :
                ajouter à la liste activation_in[c] le net activation[n]
    // création des portes OR
    pour chaque nœud n du mdd :
        a = creer_arbre_or(activation_in[n])
        relier a à activation[n]
fonction creer_arbre_or(liste) :
    tant que liste a au moins 2 éléments :
        nouvelle_liste = liste_vide
        pour chaque a et b dans liste (parcourt deux par deux) :
            o = créer une porte OR à 2 entrées
            relier a et b aux entrées de o
```

```
            ajouter la sortie de o à nouvelle_liste
        si il reste un élément a non parcouru dans liste (taille impaire) :
            ajouter a à nouvelle liste
        liste = nouvelle_liste
    retourner l'unique élément de liste
```

On peut vérifier que le circuit généré par cet algorithme respecte la sémantique du MDD spécifiée dans le chapitre précédent. Si un nœud est activé (sémantiquement), le signal d'activation du nœud défini ci-dessus est à 1 ; sinon il est à 0. On vérifie les points qui définissent la sémantique du MDD :

- Les racines sont toujours activées : elles n'ont pas de signal d'activation
- Un nœud qui n'est pas racine est activé si au moins l'un des arcs entrants est à 1, puisque le signal d'activation est généré par un arbre de portes OR. Comme les arcs entrants sont mutuellement exclusifs, un seul peut être à 1 à la fois. Réciproquement, si tous les arcs entrants sont à 0, le signal d'activation est à 0.
- Si n est un non-terminal activé et si $var(n)$ est valide, l'un de ses rails, i, est à 1. Comme le signal d'activation est à 1, la porte de Muller qui combine ce signal au rail i de $var(n)$ commute pour passer à 1, ce qui active l'arc sortant correspondant.
- Si n est une fourche activée, le 1 du signal d'activation est propagé sur tous ses arcs sortants.
- Si n est un terminal activé, le 1 du signal d'activation est propagé sur le rail $val(n)$ de $var(n)$, ce qui correspond bien à émettre une valeur.
- Si n est un non-terminal désactivé et si $var(n)$ est invalide, tous ses rails sont à 0, donc toutes les portes de Muller du nœud forcent leur sortie à 0, ce qui désactive tous ses arcs sortants.
- Si n est une fourche désactivée, le 0 du signal d'activation est propagé sur tous ses arcs sortants.
- Si n est un terminal désactivé, le 0 du signal d'activation est propagé sur le rail $val(n)$ de $var(n)$, ce qui correspond bien à une remise à zéro.

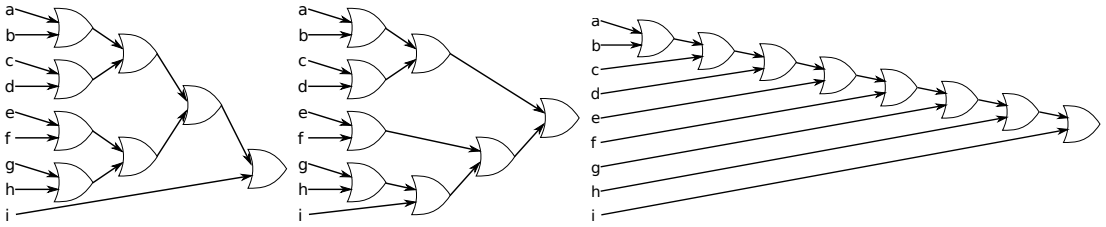
6 Probabilités et équilibrage des arbres de portes OR

6.1 Motivation

L'algorithme de synthèse en portes à deux entrées génère des arbres de portes OR, afin d'implémenter uniquement avec des portes OR à deux entrées le ou logique d'autant de rails que nécessaire. Comme le montre la figure 4.9, il y a de nombreuses façons de générer ces arbres. Cette section s'intéresse à l'optimisation de ces arbres.

L'optimisation ne doit pas ce faire au niveau du MDD, puisque le MDD spécifie uniquement l'ensemble d'arcs à regrouper par un ou logique, et ne spécifie pas plus sur la manière d'effectuer ce ou logique. Techniquement, on pourrait le faire en ajoutant dans le MDD une fourche pour chaque porte OR de l'arbre que l'on veut générer. Mais ce détournement de l'utilisation des fourches n'aurait que peu

d'intérêts. Il est beaucoup plus intéressant de faire cette optimisation directement au niveau de la synthèse, en modifiant l'algorithme *creer_ arbre_ or*.



(a) arbre de base (b) équilibré (c) totalement déséquilibré

FIG. 4.9 – Il y a plusieurs manières d'implémenter le ou logique d'un certain nombre de rails avec des portes OR à deux entrées. L'arbre généré est plus ou moins équilibré. L'arbre (a) est celui généré par l'algorithme présenté dans la section 5.

La première question est de savoir selon quel critère on souhaite optimiser l'arbre de portes. Visiblement, ce critère ne peut pas être le nombre de portes, puisque tous les arbres ont le même nombre de portes : chaque porte OR ayant deux entrées pour une sortie, elle réduit de un le nombre de signaux, il faut donc $n-1$ portes OR dans l'arbre pour n signaux, quelle que soit la structure de l'arbre.

Dès lors, les critères qui semblent intéressants à optimiser sont peu nombreux : la consommation, et le délai de passage d'une transition à travers l'arbre. Tous deux sont liés : il s'agit de minimiser en moyenne le nombre de portes traversées par une transition. Pour estimer cette moyenne, on définit la notion de probabilité de transition, qui correspond à la probabilité que le chemin participe au calcul d'un sortie : plus un chemin a une probabilité de transition élevée, plus il compte, et donc plus le nombre de portes OR qu'il traverse dans l'arbre doit être réduit.

6.2 Probabilité de transition

La probabilité de transition d'un net dans la netlist mesure la probabilité que, durant un cycle de calcul, ce net participe au calcul d'une sortie. Cette probabilité dépend des valeurs lues en entrée de la netlist : si les entrées lues ont toujours la même valeur, les chemins activés seront toujours les mêmes. La probabilité de transition d'un net est donc définie dans une netlist à partir des vecteurs de probabilité de transition des entrées de la netlist, c'est-à-dire à la probabilité que chaque valeur d'une entrée soit émise :

Définition 6.1 *Un **vecteur de probabilité des entrées** VP de la netlist est une fonction qui, à chaque entrée primaire E de la netlist, associe le vecteur d'entiers $VP(E)$ tel que pour $i \in range(E), VP(E)[i] \leq 1$ et $\sum_{i \in range(E)} VP(E)[i] \leq 1$.*

Un calcul vérifie le vecteur de probabilités VP si et seulement si pour toute entrée primaire E, sur les N_E lectures de la variable E durant le calcul, pour tout $i \in range(E)$, la valeur i est lue $VP(E)[i] \times N$ fois.

Soit un net n de la netlist, et soit VP un vecteur de probabilités des entrées.

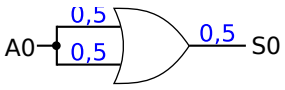
Définition 6.2 *La **probabilité de transition** de n suivant le vecteur de probabilités des entrées VP est le réel $P_{VP}(n)$ tel que, sur un calcul vérifiant le vecteur de*

probabilités VP composé de N cycles de calcul, n subit en moyenne $2P_{VP}(n) \times N$ transitions.

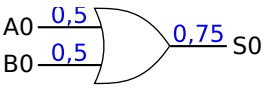
Le facteur 2 vient du fait que dans le protocole quatre phases, un signal subit deux transitions : une transition montante et une transition descendante.

6.3 Calcul des probabilités à partir du MDD

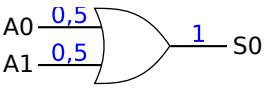
Pour un vecteur de probabilités des entrées donné, on souhaite calculer la probabilité de transition de chaque net de la netlist. Cependant, faire ce calcul sur la netlist est une tâche ardue : comme l'illustre la figure 4.10, la connaissance de la probabilité de transition des nets d'entrée d'une instance de la netlist (et de la fonction logique de cette instance) n'est pas suffisante pour calculer la probabilité de transition de la sortie de l'instance. En effet, les probabilités de transition des entrées peuvent être liées ou non, ce qui change le calcul et le résultat.



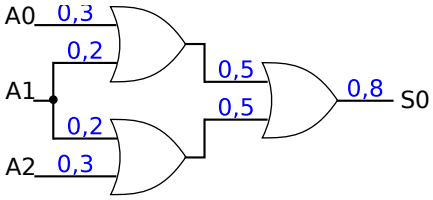
(a) fourche en entrée



(b) entrées indépendantes



(c) entrées mutuellement exclusives



(d) A_0, A_1 et A_2 sont mutuellement exclusives, les entrées de la porte de sortie sont liées. La probabilité de S_0 est la somme des probabilités de A_0, A_1 et A_2.

FIG. 4.10 – Le calcul de la probabilité de transition de la sortie d'une porte OR ne dépend pas seulement des probabilités de transition des entrées : $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$. Le tout étant de connaître $P(a \wedge b)$.

Il est donc essentiel, en plus de la probabilité de transition des entrées, de savoir si les entrées sont liées. Dans un circuit QDI, deux nets sont liés s'ils ne sont pas mutuellement exclusifs. La notion de mutuelle exclusivité joue donc un rôle majeur pour le calcul des probabilités de transition des nets. Comme la structure de MDD intègre l'information de mutuelle exclusivité entre les arcs, et que les arcs du MDD correspondent à des nets de la netlist, il semble naturel d'effectuer ce calcul à partir du MDD, plutôt qu'en considérant la netlist directement.

En première approche, il semble nécessaire de distinguer les transitions montantes et les transitions descendantes, puisque les portes logiques ne réagissent pas de la même manière aux deux types de transition. Cependant (et c'est là un autre intérêt du calcul sur le MDD plutôt que sur la netlist), les circuits que l'on considère ont une structure particulière : étant donné que chaque transition sur un net provoque une transition en sortie (puisque le circuit est QDI), et que le comportement du circuit est symétrique : après chaque phase de transitions montantes, il y a une phase de transitions descendantes, il n'est pas nécessaire de distinguer les transitions

montantes et descendantes. Il suffit de faire les calculs sur l'activation des arcs du MDD, qui correspond à une transition montante du net correspondant : on sait que l'arc sera désactivé, ce qui causera une transition descendante sur le même net.

Dans un MDD, il y a un seul type de convergence possible : un ensemble d'arcs mutuellement exclusifs converge vers un nœud, et l'activation de ce nœud est le «ou logique» de cet ensemble d'arcs. Étant donné que l'ensemble d'arcs est mutuellement exclusif, il est possible de calculer la probabilité d'activation du nœud (c'est-à-dire la probabilité de transition du signal d'activation du nœud) : c'est la somme des probabilités de transition des arcs convergeant.

Afin de pouvoir calculer les probabilités de transition sur le MDD de manière locale, il est nécessaire d'ajouter des hypothèses sur le MDD :

- les entrées du circuit sont indépendantes : la connaissance de la valeur d'une variable d'entrée ne modifie pas les probabilités de transition des rails des autres variables.
  Cette hypothèse est nécessaire puisqu'il faut avoir une information sur les interaction entre les variables pour pouvoir calculer les probabilités de sortie des nœuds combinant ces variables. Cette hypothèse n'est pas forcément vérifiée : on peut concevoir un circuit tel que deux entrées ne puissent être présentes en même temps, par exemple. De tels circuits sont ignorés ici.
- le MDD satisfait la propriété d'unicité : sur chaque chemin, il existe au plus un nœud étiqueté par une variable donnée.
  Cette propriété permet de s'assurer que deux nœuds sur un même chemin ont bien leurs variables indépendantes : même si les variables sont indépendantes deux à deux, si deux nœuds sont étiquetés par la même variable, il y a dépendance...

Le calcul de la probabilité de transition des arcs sortants d'une fourche est immédiat : les arcs ont tous la même probabilité, qui correspond à la probabilité d'activation de la fourche. Pour les non-terminaux, le calcul est légèrement plus complexe : chaque rail de la variable du nœud est combiné par une porte de Muller au signal d'activation. Ce signal d'activation dépend des variables précédemment lues dans le chemin ; il est donc indépendant de la variable (par hypothèse). Il convient donc de multiplier la probabilité d'activation par la probabilité de transition du rail, pour chaque arc sortant.

Algorithme 6.1

```
fonction calcul_proba_mdd(mdd, VP) :
    // initialisation
    pour chaque nœud n du mdd :
        si n est une racine :
            proba_activation[n] = 1
        sinon:
            proba_activation[n] = 0
    // calcul
    pour chaque nœud n du mdd, parcours en largeur d'abord :
        si n est une fourche :
            pour chaque nœud c ∈ childs(n) :
                proba_activation[c] += proba_activation[n]
        si n est un non-terminal :
```

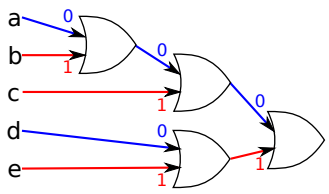
pour chaque $i \in range(var(n))$:
$p = proba\_activation[n] \times VP(var(n))[i]$
$proba\_activation[child(i,\ n)] += p$

6.4 Algorithme de génération de l'arbre optimisé

Maintenant que la probabilité de transition de chaque net d'entrée des arbres de OR est connue, on souhaite générer l'arbre optimal, en fonction de ces probabilités.

Afin de parler d'arbre optimal, il est nécessaire de définir le critère suivant lequel on souhaite optimiser l'arbre. Ce que l'on veut, c'est qu'en moyenne, le chemin actif traverse le plus petit nombre de portes possible, c'est-à-dire que la moyenne de la longueur des chemins de l'arbre pondérée par la probabilité de transition soit minimale.

Ce problème est équivalent au problème bien connu de compression de fichier par codage selon un nombre de bits variable, comme le montre la figure 4.11. Chaque rail d'entrée de l'arbre de portes OR correspond à un caractère du fichier à compresser. La probabilité de transition du rail correspond à la probabilité d'occurrence du caractère. L'arbre lui-même correspond au codage choisi : c'est un arbre binaire, chaque chemin de la racine à une feuille peut être coder en binaire, 0 correspondant au choix du fils gauche et 1 au choix du fils droit, ce qui donne la manière de coder le caractère correspondant. La trace d'exécution d'un calcul dans le circuit, c'est-à-dire la séquence de transitions d'un rail d'entrée de l'arbre vers sa sortie, codée de la même manière, c'est-à-dire 0 si la transition provient du fils gauche, et 1 si elle provient du fils droit, correspond au fichier compressé (modulo le fait que le codage part de la racine aux feuilles, alors que les transitions elles remontent l'arbre depuis les feuilles aux racines ; ceci ne modifie pas le nombre de portes traversées).



(a) arbre de portes OR

caractère	codage
a	000
b	001
c	01
d	10
e	11

(b) codage

FIG. 4.11 – Un arbre de portes OR est analogue à un codage selon un nombre de bits variable.

Or, c'est bien la taille de cette trace d'exécution que l'on souhaite minimiser, puisqu'elle correspond au nombre de portes activées, qui participent à la fois au délai du circuit et à sa consommation dynamique. Tout comme, dans le problème de compression de fichier, on souhaite minimiser la taille du fichier compressé.

Ce problème se résout donc par l'algorithme de Huffman [43], qui construit directement l'arbre. Cet algorithme consiste, à partir d'une liste d'objets étiquetés par une probabilité, de fusionner les deux objets de la liste ayant la priorité la plus faible, pour obtenir un nouvel objet ayant la probabilité somme des deux, qui est réinséré dans la liste. L'algorithme termine lorsqu'il ne reste plus qu'un seul objet.

Algorithme 6.2

fonction arbre_ Huffman(*liste*) :
    tant que *liste contient plusieurs éléments :*
        a = *retirer_plus_petit_élément*(*liste*)
        b = *retirer_plus_petit_élément*(*liste*)
        c = *fusionner*(a, b)
        $proba(c) = proba(a) + proba(b)$
        ajouter c *dans liste avec pour probabilité* $proba(c)$
    retourner *l'unique élément de liste*

Dans notre cas, fusionner deux rails consiste à les regrouper par une porte OR ; le résultat de la fusion est le rail sortant de la porte OR. La figure 4.12 montre un exemple d'arbre obtenu avec l'algorithme de Huffman.

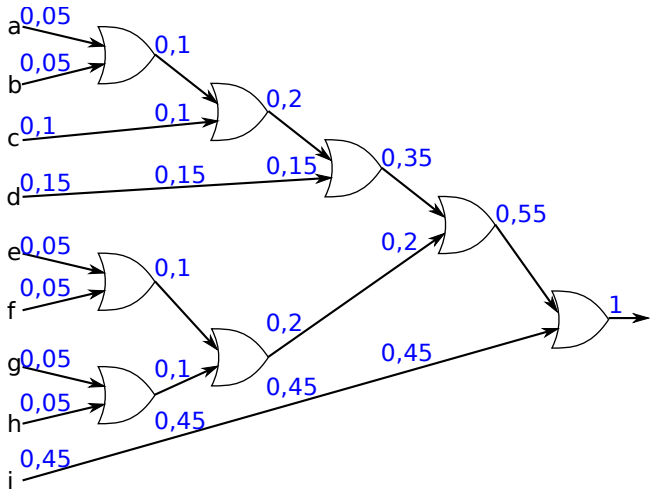


FIG. 4.12 – exemple d'arbre obtenu avec l'algorithme de Huffman.

7 Conclusion

Dans ce chapitre, on a mis en place le flot de synthèse, qui est conçu de manière à synthétiser des circuits quasi insensibles aux délais. Pour cela, on a mis en place des algorithmes d'optimisation de MDD, en prouvant que ces optimisations préservent la sémantique du MDD. Puisque ces optimisations ont lieu avant la synthèse, et qu'elles préservent la sémantique du MDD, elles ne compromettent pas le fait que le circuit synthétisé sera quasi insensible aux délais.

Concernant l'étape de synthèse proprement dite, deux algorithmes sont développés. Le premier, le plus simple, réalise une synthèse déjà connue sous le nom de DIMS. Les circuits synthétisés ainsi sont énormes, sauf pour un très petit nombre d'entrée. Mais l'algorithme est très simple, et il sert de brique de base pour la preuve formelle de quasi insensibilité aux délais du chapitre suivant. Le second algorithme permet de synthétisé les MDD sous forme de circuits n'employant que des portes à deux entrées. Cette limitation du fan-in est importante, puisque, passé le stade de la synthèse, on ne peut effectuer que des transformations qui préservent la quasi

insensibilité aux délais du circuit, qui a été prouvée pour le circuit issu de la synthèse. Ainsi, cette limitation du fan-in permet d'effectuer ensuite une projection technologique du circuit sans avoir à effectuer de décomposition, ce qui ne serait pas autorisé.

Pour finir, on a mis en place un mécanisme de calcul de la probabilité de transition des nets. La connaissance des probabilités de transition permet d'équilibrer automatiquement les arbres de portes OR, et ainsi d'optimiser vitesse et consommation du circuit.

Chapitre 5

Correction formelle du flot de synthèse

1 Introduction

Dans ce chapitre, on démontre l'un des points cruciaux de cette thèse : garantir qu'un circuit issu de la synthèse d'un MDD est quasi insensible aux délais.

Le flot de synthèse développé repose sur cette preuve pour garantir que les circuits générés sont quasi insensibles aux délais : aucune vérification de la quasi insensibilité aux délais n'est effectuée durant la synthèse, une telle vérification étant trop coûteuse.

Ce travail a fait l'objet d'une publication en juin 2004 [44]. Cette publication ne concerne que la synthèse DIMS ; la synthèse en portes deux entrées n'avait pas encore été développée à cette date. Dans ce chapitre, on considère aussi la synthèse en portes deux entrées, en se basant sur le résultat obtenu dans le cas de la synthèse DIMS.

2 Caractérisation formelle de la quasi insensibilité aux délais

Afin de montrer qu'un circuit, modélisé par un ensemble de règles de production, est quasi insensible aux délais, il est nécessaire d'avoir une définition formelle de cette propriété. L'article [8] donne une condition nécessaire et suffisante pour qu'un ensemble de règles de production soit quasi insensible aux délais. Tout d'abord, donnons quelques définitions.

Définition 2.1 *Le **résultat** d'une transition est un prédicat, noté R, tel que $R(x \nearrow) = x$ et $R(x \searrow) = \overline{x}$.*

Définition 2.2 *Un couple de règles de production $B^+ \rightarrow x \nearrow$ et $B^- \rightarrow x \searrow$ est **non-interférant** durant un calcul si et seulement si $B^+ \wedge B^-$ est toujours faux durant le calcul.*

Un ensemble de règles de production est non-interférant si et seulement si chaque couple de règles de production est non-interférant.

Théorème 2.1 *Un couple de règles de production combinatoire est toujours non-interférant.*

En effet, si un couple de règles de production $B^+ \rightarrow x \nearrow, B^- \rightarrow x \searrow$ est combinatoire, alors $B^+ = \overline{B^-}$, et donc $B^+ \wedge B^- = \overline{B^-} \wedge B^-$ est toujours faux.

Définition 2.3 *Une règle de production $G \rightarrow t$ est **stable** durant un calcul si et seulement si G ne peut passer de vrai à faux que dans les états du calcul pour lesquels $R(t)$ est vrai.*

Un ensemble de règles de production est stable si et seulement si chaque règle de production est stable.

Ces définitions permettent de caractériser la quasi insensibilité aux délais d'un circuit :

Théorème 2.2 *Un circuit est **quasi insensible aux délais** si et seulement si l'ensemble de règles de production qui le décrit est stable et non-interférant.*

En effet, si l'ensemble de règles de production n'est pas stable, il existe une règle de production $B \rightarrow t$ non stable. Donc, il existe un état dans lequel $\overline{R}(t) \wedge B$ est vraie, suivi par un état dans lequel $\overline{R}(t) \wedge \overline{B}$ est vraie. Ainsi, la variable x de la transition t peut subir un aléa : dans le premier état, puisque B est vraie, la règle de production est activée, et donc x subit une transition. Mais avant que x subisse cette transition, B devient fausse. Ainsi, le comportement n'est pas spécifié, un aléa peut se produire, et le circuit n'est donc pas QDI. Si un couple de règles de production est interférant, il existe un état où les deux gardes sont vraies, le comportement n'est donc pas spécifié, et le circuit n'est pas non plus QDI.

Réciproquement, supposons que l'ensemble de règles de production est stable et non-interférant. Soit un couple de règles de production $B^+ \rightarrow x \nearrow$ et $B^- \rightarrow x \searrow$. Puisque les règles de production sont stables, on sait que si B^+ est vraie, alors cette garde reste vraie jusqu'à ce que $x = 1$, et de même pour la garde B^- : la porte ne peut donc pas subir d'aléa.

3 La synthèse DIMS génère des circuits QDI

On démontre dans cette section qu'un circuit issu de l'algorithme de synthèse DIMS, présenté dans le chapitre précédent, est forcément quasi insensible aux délais.

Pour cela, il faut tout d'abord modéliser ce circuit sous forme d'un ensemble de règles de production. Puis, quelques hypothèses doivent être faites sur le MDD. Enfin, on prouve que l'ensemble de règles de production est non-interférant et stable, et qu'il est donc quasi insensible aux délais en vertu du théorème 2.2.

3.1 Génération des règles de production à partir du MDD

Puisque la caractérisation de la quasi insensibilité aux délais d'un circuit passe par sa description sous forme de règles de production, il est nécessaire de construire un ensemble de règles de production à partir du circuit issu d'un MDD. En fait, cet

ensemble de règles de production peut être construit directement à partir du MDD, par un algorithme similaire à l'algorithme de synthèse du MDD.

De plus, le modèle de circuit utilisé dans la caractérisation de la quasi insensibilité aux délais est fermé, c'est-à-dire que les circuits considérés n'ont pas de ports primaires. Cependant, nos circuits sont ouverts. Pour les rendre fermés, il suffit de modéliser l'environnement à l'aide de règles de production, qui décrivent le comportement de l'environnement, c'est-à-dire implémentent le protocole quatre phases, sur les ports primaires du circuit, comme l'illustre la figure 5.1.

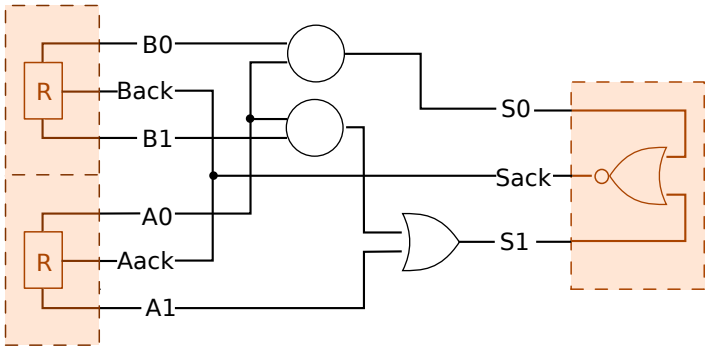


FIG. 5.1 – Fermer un circuit revient à modéliser l'environnement au niveau des ports primaires. Les blocs R choisissent la valeur à émettre, c'est-à-dire le rail activé par l'environnement lorsque l'acquittement est reçu.

Règles de production du MDD

Considérons chaque terminal f du MDD. Pour ce terminal f, on note $C^f = C_0^f, C_1^f, \ldots, C_{p-1}^f$ l'ensemble des chemins du MDD d'une racine à f. Considérons l'un de ces chemins, $C_i^f, 0 \leq i < p$. On note $t_{i,0}^f, t_{i,1}^f, \ldots, t_{i,k_i}^f$ les tronçons de ce chemin jusqu'au terminal f, non inclus.

La synthèse DIMS construit une porte de Muller par chemin. On doit donc construire un couple de règles de production pour ce chemin. Pour cela, on définit le rail d'un tronçon t, de la manière suivante. On rappelle que si $n = node(t)$ est le nœud du tronçon t (c'est-à-dire le nœud du MDD qui correspond au tronçon t dans le chemin considéré), que $var(n)$ est la variable qui l'étiquette (c'est-à-dire la variable qui est lue), et que $val(t)$ est la valeur qui doit être lue sur $var(n)$ pour que l'on traverse t (si c'est une autre valeur, c'est un autre chemin qui s'activera).

- si $node(t)$ est un non-terminal, $rail(t) = var(node(t))_{val(t)}$ est le rail numéro $val(t)$ de la variable $var(node(t))$.
- si $node(t)$ est une fourche, $rail(t) = true$.
- si $node(t)$ est un terminal, $rail(t) = var(node(t))_{val(node(t))}$ est le rail $val(node(t))$ de la variable $var(node(t))$.

Ainsi, le couple de règles de production modélisant le chemin C_i^f se construit sous la forme suivante :

$$\begin{aligned} rail(t_{i,0}^f) \wedge rail(t_{i,1}^f) \wedge \ldots \wedge rail(t_{i,k_i}^f) &\rightarrow C_i^f \nearrow \\ \overline{rail(t_{i,0}^f)} \wedge \overline{rail(t_{i,1}^f)} \wedge \ldots \wedge \overline{rail(t_{i,k_i}^f)} &\rightarrow C_i^f \searrow \end{aligned}$$

De plus, pour chaque terminal, l'algorithme DIMS génère une porte OR, qui regroupe les chemins arrivant à cette porte. Ainsi, le couple de règles de production modélisant la porte OR correspondant au terminal f se construit sous la forme suivante :

$$\begin{aligned} C_0^f \vee C_1^f \vee \ldots \vee C_p^f &\rightarrow rail(f) \nearrow \\ \overline{C_0^f} \wedge \overline{C_1^f} \wedge \ldots \wedge \overline{C_p^f} &\rightarrow rail(f) \searrow \end{aligned}$$

Règles de production de l'environnement

Soit S une sortie primaire du circuit. L'environnement implémente le protocole quatre phases sur cette sortie, c'est-à-dire que :

- Si l'un des rails de S passe à 1, l'acquittement passe à 0.
- Si le rail de S qui était à 1 passe à 0 (et donc que tous les rails de S sont à 0), l'acquittement passe à 1.

L'environnement peut donc être modélisé par le couple de règles de production suivant :

$$\begin{aligned} \bigvee_{i \in range(S)} S_i &\rightarrow S_{ack} \searrow \\ \bigwedge_{i \in range(S)} \overline{S_i} &\rightarrow S_{ack} \nearrow \end{aligned}$$

Soit A une entrée primaire du circuit.

- Si l'acquittement de A passe à 1, l'un des rails $i \in range(a)$ de A passe à 1. Ce rail est choisi par l'environnement, on considère donc qu'il est aléatoire (et on ne tient donc pas compte du fait qu'il peut y avoir des hypothèses supplémentaires que ferait le circuit).
- Si l'acquittement de A passe à 0, le rail i de A passe à 0.

L'environnement peut donc être modélisé par le couple de règles de production suivant :

$$\begin{aligned} A_{ack} &\rightarrow A_i \nearrow \\ \overline{A_{ack}} &\rightarrow A_i \searrow \end{aligned}$$

3.2 Conditions sur le MDD

Afin de démontrer que le circuit synthétisé est quasi insensible aux délais, il est nécessaire d'énoncer des conditions que le MDD doit respecter.

Tout d'abord, le MDD doit être bien formé. En effet, on a vu au chapitre 3 qu'un MDD qui n'est pas bien formé ne peut pas générer un circuit QDI.

De plus, il est nécessaire d'ajouter des conditions sur la relation entre les acquittements des entrées et des sorties. Pour cela on ajoute la condition que le circuit est correctement acquitté. La notion d'acquittement correct est définie de la manière suivante :

Soient A une entrée et S une sortie du circuit.

Définition 3.1 *Le couple (A, S) est* ***correctement acquitté*** *si et seulement si, lorsque A est consommée lors de la production de S, A_{ack} ne peut passer à 0 avant que S_{ack} ne soit à 0, et A_{ack} ne peut passer à 1 avant que S_{ack} ne soit à 1.*

Le circuit est ***correctement acquitté*** *si tous les couples (entrée, sortie) sont correctement acquittés.*

On démontre alors le théorème suivant :

Théorème 3.1 *Un MDD bien formé modélisant un circuit correctement acquitté génère via synthèse DIMS un ensemble de règles de productions* ***quasi insensible aux délais****.*

3.3 Non-interférence

On démontre qu'un ensemble de règles de production issu d'un MDD est non-interférant. Pour cela, on considère les différents types de couples de règles de production.

Soit un terminal f et un chemin C_i^f allant d'une racine du MDD à f. En reprenant les notations précédentes, le couple de règles de production généré à partir de ce chemin est :

$$\begin{aligned} rail(t_{i,0}^f) \wedge rail(t_{i,1}^f) \wedge \ldots \wedge rail(t_{i,k_i}^f) &\rightarrow C_i^f \nearrow \\ \overline{rail(t_{i,0}^f)} \wedge \overline{rail(t_{i,1}^f)} \wedge \ldots \wedge \overline{rail(t_{i,k_i}^f)} &\rightarrow C_i^f \searrow \end{aligned}$$

Ce couple de règles de production est non-interférant si et seulement si la fonction suivante est toujours fausse :

$$rail(t_{i,0}^f) \wedge rail(t_{i,1}^f) \wedge \ldots \wedge rail(t_{i,k_i}^f) \wedge \overline{rail(t_{i,0}^f)} \wedge \overline{rail(t_{i,1}^f)} \wedge \ldots \wedge \overline{rail(t_{i,k_i}^f)}$$

Ce qui est bien le cas, de manière évidente (la fonction est de la forme $A \wedge \overline{A} \wedge \ldots$).

Les autres règles de production (sorties, environnement) sont combinatoires, donc non-interférantes.

3.4 Stabilité

On démontre qu'un ensemble de règles de production issu d'un MDD est non-interférant.

Pour cela, on commence par montrer le lemme suivant : Soit un terminal f du MDD. Soient C_i^f les chemins du MDD reliant une racine au terminal f. On considère le circuit issu de la synthèse DIMS du MDD.

Lemme 3.1 *Le circuit implémente le protocole quatre phases, et ainsi une boucle de causalité de transitions :*

1. *$rail(f)$ ne peut passer à 1 avant que $C^f_{i_0}$, l'un des chemins, ne soit à 1.*
2. *Le chemin $C^f_{i_0}$ ne peut passer à 1 avant que, pour chaque tronçon $t \in C^f_{i_0}$, $rail(t)$ ne soit à 1.*
3. *Pour chaque tronçon $t \in C^f_{i_0}$, $rail(t)$ ne peut passer à 1 avant que $var(t)_{ack}$ ne soit à 1.*
4. *Pour chaque tronçon $t \in C^f_{i_0}$, $var(t)_{ack}$ ne peut passer à 1 avant que $var(f)_{ack}$ ne soit à 1.*
5. *$var(f)_{ack}$ ne peut passer à 1 avant que tous les rails de $var(f)$ ne soient à 0 (en particulier $rail(f)$).*
6. *$rail(f)$ ne peut passer à 0 avant que tous les C^f_i ne soient à 0. Comme les chemins sont mutuellement exclusifs, un seul pouvait être à 1, c'est $C^f_{i_0}$. Il suffit donc que $C^f_{i_0}$ soit à 0.*
7. *Le chemin $C^f_{i_0}$ ne peut passer à 0 avant que, pour chaque tronçon $t \in C^f_{i_0}$, $rail(t)$ ne soit à 0.*
8. *Pour chaque tronçon $t \in C^f_{i_0}$, $rail(t)$ ne peut passer à 0 avant que $var(t)_{ack}$ ne soit à 0.*
9. *Pour chaque $t \in C^f_{i_0}$, $var(t)_{ack}$ ne peut passer à 0 avant que $var(f)_{ack}$ ne soit à 0.*
10. *$var(f)_{ack}$ ne peut passer à 0 avant que $rail(f)$ ne soit à 1.*

La figure 5.2 illustre l'enchaînement de ces étapes, qui se fait dans le sens opposé aux transitions : l'intérêt est de remonter la chaîne de causalité des transitions, pour montrer qu'une transition a été causée par une transition précédente.

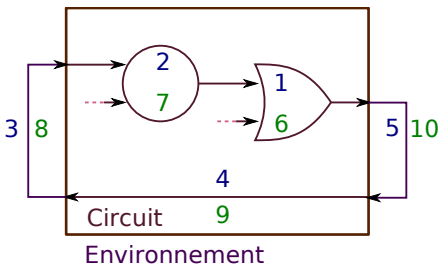


FIG. 5.2 – Les étapes du lemme 3.1 remontent la chaîne de causalité des transitions du circuit.

La plupart des étapes de ce lemme sont en fait une simple transcription des règles de production : une transition ne peut arriver avant que sa garde ne soit vraie. Les étapes 4 et 9 utilisent l'hypothèse selon laquelle le circuit est correctement acquitté.

Chemins

Le lemme 3.1 permet de montrer que les règles de production des chemins sont stables. Soit une telle règle, de la forme suivante :

$$rail(t^f_{i,0}) \wedge rail(t^f_{i,1}) \wedge \ldots \wedge rail(t^f_{i,k_i}) \;\rightarrow\; C^f_i \nearrow$$

Soit un tronçon $t^f_{i,j} \in C^f_i$. Le lemme 3.1 montre que $rail(t^f_{i,j})$ ne peut passer à 0 avant que C^f_i ne soit à 1 (en faisant un demi-cycle, c'est-à-dire en utilisant les étapes 8, 9, 10, 1). Ainsi, la garde de la règle de production ne peut devenir fausse avant que la transition ait eu lieu, la règle de production est donc stable.

De même, si l'on considère une règle de la forme suivante :

$$\overline{rail(t^f_{i,0})} \wedge \overline{rail(t^f_{i,1})} \wedge \ldots \wedge \overline{rail(t^f_{i,k_i})} \;\; \rightarrow \;\; C^f_i \searrow$$

D'après le lemme, $rail(t^k_{i,j})$ ne peut passer à 1 avant que C^f_i ne soit à 0 (étapes 3, 4, 5, 6). La règle de production est stable.

Sorties

De même, les règles de production des sorties sont stables. Soit une règle de la forme suivante :

$$C^f_0 \vee C^f_1 \vee \ldots \vee C^f_p \;\; \rightarrow \;\; rail(f) \nearrow$$

D'après le lemme, chaque chemin C^f_i ne peut passer à 0 avant que $rail(f)$ ne soit à 1 (étapes 7, 8, 9, 10), donc la règle de production est stable.

De même, pour une règle de production de la forme suivante :

$$\overline{C^f_0} \wedge \overline{C^f_1} \wedge \ldots \wedge \overline{C^f_p} \;\; \rightarrow \;\; rail(f) \searrow$$

D'après le lemme, un chemin C^f_i ne peut passer à 1 avant que $rail(f)$ ne soit à 0 (étapes 2, 3, 4, 5).

Environnement

De même, les règles de production de l'environnement sont stables. Soit une règle de la forme suivante :

$$\bigvee_{i \in range(S)} S_i \;\; \rightarrow \;\; S_{ack} \searrow$$

La garde ne peut devenir fausse que si les rails de S ne passe à 0. Chaque rail $i \in range(S)$ correspond à un terminal f_i (tel que $var(f_i) = S$ et $val(f_i) = i$). D'après le lemme, $rail(f_i)$ ne peut passer à 0 avant que $rail(f_i)_{ack}$ ne passe à 0 (étapes 6, 7, 8, 9), c'est-à-dire avant que S_{ack} ne passe à 0. La règle de production est donc stable.

De même, pour une règle de production de la forme suivante :

$$\bigwedge_{i \in range(S)} \overline{S_i} \;\; \rightarrow \;\; S_{ack} \nearrow$$

La garde ne peut devenir fausse que si l'un des rails de S ne passe à 1. Ce rail correspond à un terminal f. D'après le lemme, $rail(f)$ ne peut passer à 1 avant que

$var(f)_{ack}$ ne passe à 1 (étapes 1, 2, 3, 4), c'est-à-dire avant que S_{ack} ne passe à 1. La règle de production est donc stable.

De même, pour une règle de production de la forme suivante :

$$A_{ack} \rightarrow A_{i_0} \nearrow$$

$i_0 \in range(A)$ est la valeur que doit prendre A dans le cycle de calcul considéré. On considère un tronçon t_{i_0} tel que $var(t_{i_0}) = A$ et $val(t_{i_0}) = i_0$. On a donc $var(t_{i_0})_{ack} = A_{ack}$. D'après le lemme, $var(t_{i_0})_{ack}$ ne peut passer à 0 avant que $rail(t_{i_0})$ ne passe à 1 (étapes 9, 10, 1, 2), c'est-à-dire avant que A_{i_0} ne passe à 1. La règle de production est donc stable.

De même, pour une règle de production de la forme suivante :

$$\overline{A_{ack}} \rightarrow A_{i_0} \searrow$$

D'après le lemme, $var(t_{i_0})_{ack}$ ne peut passer à 1 avant que $rail(t_{i_0})$ ne passe à 0, c'est-à-dire avant que A_{i_0} ne passe à 0 (étapes 4, 5, 6, 7). La règle de production est donc stable.

On a montré que l'ensemble de règles de production était stable et non-interférant. D'après le théorème 2.2, il est donc quasi insensible aux délais.

4 La synthèse en porte 2 entrées génère des circuits QDI

On souhaite maintenant démontrer le théorème suivant :

Théorème 4.1 *Un MDD bien formé modélisant un circuit correctement acquitté génère via synthèse en portes à deux entrées un ensemble de règles de productions* ***quasi insensible aux délais.***

4.1 Règles de production

Soit un non-terminal n du MDD. À partir de ce nœud, on génère une porte OR, regroupant l'ensemble des arcs entrants, et $|range(var(n))|$ portes de Muller à deux entrées.

La porte OR se modélise avec les règles de production suivantes (on assimile les éléments de $ein(n)$, qui sont les arcs entrants de n, avec les nets de la netlist associés ; $act(n)$ est le signal d'activation de n). :

$$\bigvee_{e \in ein(n)} e \rightarrow act(n) \nearrow$$

$$\bigwedge_{e \in ein(n)} \overline{e} \rightarrow act(n) \searrow$$

Pour $i \in range(var(n))$, la porte de Muller correspondant à la valeur i se modélise avec les règles de production suivantes ($eout(i,n)$ est l'arc sortant de n correspondant à la valeur i) :

$$\begin{array}{rcl} act(n) \wedge var(n)_i & \rightarrow & eout(i,n) \nearrow \\ \overline{act(n)} \wedge \overline{var(n)_i} & \rightarrow & eout(i,n) \searrow \end{array}$$

Soit maintenant une fourche n du MDD. À partir de cette fourche, on génère une porte OR, qui regroupe l'ensemble des arcs sortants. Cette porte OR est modélisée par les mêmes règles de production que dans le cas d'un non-terminal. Le signal d'activation de la fourche est propagé à chaque arc sortant de la fourche : Soit $e \in eout(n)$.

$$\begin{array}{rcl} act(n) & \rightarrow & e \nearrow \\ \overline{act(n)} & \rightarrow & e \searrow \end{array}$$

4.2 Similitude avec la synthèse DIMS

Afin de simplifier la preuve de quasi insensibilité aux délais du circuit issu d'une synthèse en portes deux entrées, on souhaite se ramener au cas de la synthèse DIMS, traité ci-dessus.

La remarque est la suivante : synthétiser un MDD en portes deux entrées revient à le découper en un ensemble de MDDs de profondeur 2, et à synthétiser chaque petit MDD avec l'algorithme DIMS, comme l'illustre la figure 5.3.

Ainsi, au lieu de considérer le MDD, et d'analyser les règles de production issues de sa synthèse en portes deux entrées, on considère un ensemble de MDDs de profondeur 2 qui s'assemblent pour former le MDD initial, et on se ramène à la synthèse DIMS que l'on a étudiée ci-dessus. Ce qui apparaît de nouveau par rapport au cas de la synthèse DIMS, c'est que l'on n'a plus un seul MDD, mais un ensemble de MDDs qui s'assemblent, à l'aide de variables internes (c'est-à-dire de variables qui ne sont pas des ports primaires du circuit, comme la variable M dans la figure 5.3.

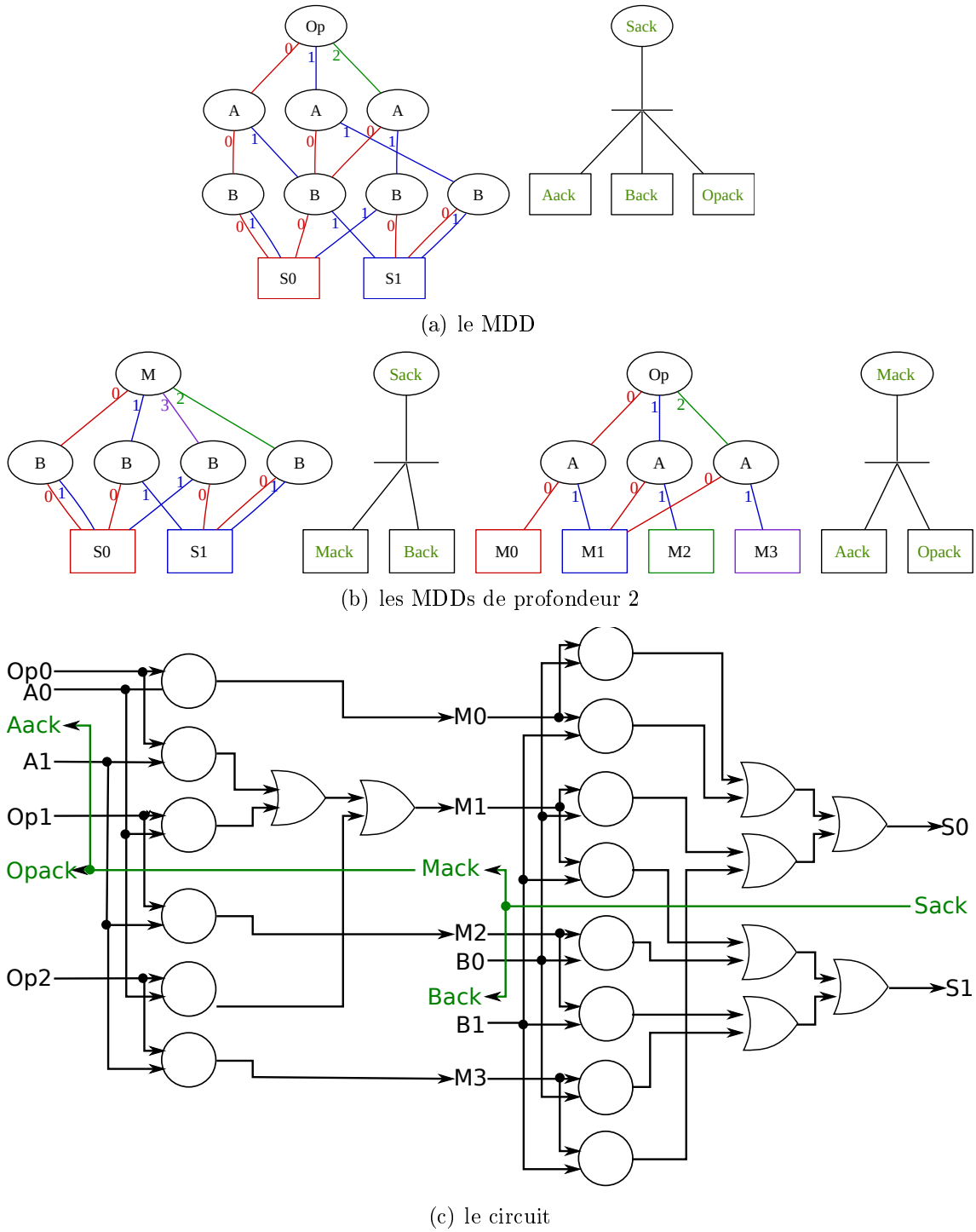
4.3 Non-interférence

Les règles de production issues d'une synthèse DIMS sont toujours non-interférantes, indépendemment du circuit global, comme on l'a montré dans la section précédente. Il n'y a donc rien à faire ici.

4.4 Stabilité

Pour montrer que l'ensemble de règles de production considéré est stable, il suffit de montrer que le lemme 3.1 est toujours vrai dans le cas d'un ensemble de MDDs qui s'assemblent.

Les étapes 1, 2, 4, 6, 7 et 9 sont des retranscriptions des règles de production générées, et sont donc toujours vraies, indépendemment du fait que l'on puisse maintenant considérer une variable interne.



(a) le MDD

(b) les MDDs de profondeur 2

(c) le circuit

FIG. 5.3 – Le circuit issu de la synthèse en portes deux entrées d'un MDD (a) est le même que celui issu de la synthèse DIMS d'un ensemble de MDDs de profondeur 2 (b), qui s'assemblent pour former le circuit (c).

Considérons les étapes 3 et 8. Soit une feuille f. On veut montrer que pour chaque tronçon t du chemin considéré $C^f_{i_0}$, $rail(t)$ ne peut passer à 1 avant que $var(t)_{ack}$ ne soit à 1, et $rail(t)$ ne peut passer à 0 avant que $var(t)_{ack}$ ne soit à 0.

Si $var(t)$ est une entrée primaire du circuit, cette proposition est immédiatement

vraie, puisque l'environnement implémente le protocole quatre phases sur $var(t)$. Il faut donc considérer le cas où $var(t)$ est une variable interne.

Dans ce cas, il existe une feuille f' telle que $var(f') = var(t)$ et $val(f') = val(t)$. Considérons un chemin $C_i^{f'}$. Supposons que tous les tronçons du chemin ont pour variable des ports primaires.

On peut alors appliquer les étapes 1, 2, 3, 4 du lemme, pour $f' : rail(f') = rail(t)$ ne peut passer à 1 avant que $var(f')_{ack} = var(t)_{ack}$ ne soit à 1.

On peut aussi appliquer les étapes 6, 7, 8, 9 du lemme, pour $f' : rail(f') = rail(t)$ ne peut passer à 0 avant que $var(f')_{ack} = var(t)_{ack}$ ne soit à 0.

On a donc montré les étapes 3 et 8 du lemme pour f.

Si l'un des tronçons du chemin a pour variable une variable interne, il suffit de faire le même raisonnement : par récurrence sur le nombre de variables internes qu'il faut traverser, on montre que les étapes 3 et 8 sont vraies. Le nombre de variables qu'il faut traverser est forcément fini, puisque l'ensemble de MDD est issu de la décomposition d'un seul MDD : traverser les variables internes correspond à parcourir les chemins du MDD d'origine.

Considérons maintenant les étapes 5 et 10. Soit une feuille f. On veut montrer que $var(f)_{ack}$ ne peut passer à 1 avant que tous les rails de $var(f)$ ne soient à 0, et que $var(f)_{ack}$ ne peut passer à 0 avant que $rail(f)$ ne soit à 1.

Comme ci-dessus, si $var(f)$ est une sortie primaire du circuit, cette proposition est immédiate. On considère donc le cas où $var(f)$ est une variable interne.

On considère alors un tronçon t' tel que $var(t') = var(f)$. Ce tronçon appartient au chemin $C_i^{f'}$, qui arrive sur le terminal f'. Supposons que $var(f')$ est une sortie primaire du circuit.

On peut alors appliquer les étapes 4, 5, 6, 7 du lemme, pour $f' : var(t')_{ack} = var(f)_{ack}$ ne peut passer à 1 avant que $rail(t') = rail(f)$ ne soit à 0. Ce raisonnement est vrai pour tout t' tel que $var(t') = var(f)$, donc $var(f)_{ack}$ ne peut passer à 1 avant que tous les rails de $var(f)$ ne soient à 0.

On peut aussi appliquer les étapes 9, 10, 1, 2 du lemme, pour $f' : var(t')_{ack} = var(f)_{ack}$ ne peut passer à 0 avant que $rail(t)$ ne soit à 1.

On a donc montré les étapes 5 et 10 du lemme.

Comme ci-dessus, si $var(f')$ est une variable interne, il suffit de faire le même raisonnement. On montre par récurrence sur le nombre de variables internes à traverser que les étapes 5 et 10 sont vraies.

5 Étapes de transformation

On souhaite maintenant montrer que les différentes transformations que l'on est amené à effectuer préservent la quasi insensibilité aux délais du circuit.

5.1 Optimisations du MDD

Les étapes d'optimisation, qui sont toutes effectuées sur le MDD, ne peuvent pas modifier la quasi insensibilité aux délais du circuit, puisqu'elles sont effectuées avant la synthèse. Par contre, elles peuvent modifier les propriétés du MDD sur lesquelles

repose la preuve de quasi insensibilité aux délais : le MDD doit toujours être bien formé, et correctement acquitté.

5.2 Projection technologique : fusion de portes

Lors de la projection technologique, certaines portes sont fusionnées. On montre ici que la fusion de portes ne modifie pas la quasi insensibilité aux délais du circuit.

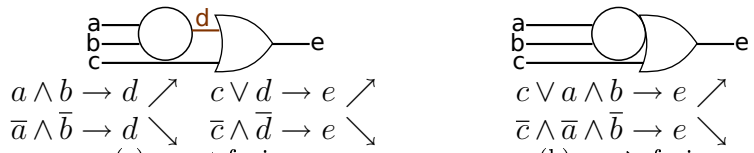
Restrictions sur la fusion

On veut définir la fusion de deux portes au niveau des règles de production. Tout d'abord, il faut définir quelles règles de production on peut fusionner.

On ne peut fusionner deux portes que si elles sont reliées, c'est-à-dire si la sortie de la première est une entrée de la seconde. Au niveau des règles de production, cela revient à ce que les gardes de la seconde dépendent de la variable de la première.

On ne peut pas fusionner deux portes si le net qui relie les deux est connecté à d'autres portes, étant donné que ce net va disparaître dans la fusion. Il ne faut donc qu'aucune autre règle de production ne dépende de la variable qui va disparaître.

De plus, une fusion ne peut faire disparaître un net s'il est mémorisant, c'est-à-dire s'il est produit par une porte qui n'est pas combinatoire. En effet, dans ce cas, le résultat de la fusion ne peut être modélisé par un seul couple de règles de production, puisque cela reviendrait à transférer la mémoire interne en sortie, et le comportement serait différent, comme l'illustre la figure 5.4 : si l'on fusionne les deux portes de la figure 5.4(a) en une seule porte, modélisée par un seul couple de règles de production figure 5.4(b), alors la séquence $a \searrow b \searrow a \nearrow c \searrow$ cause une transition descendante en sortie de la figure 5.4(a), mais pas de transition en sortie de la figure 5.4(b)



(a) avant fusion (b) après fusion

FIG. 5.4 – Le résultat de la fusion des portes n'est pas modélisable par un couple de règles de production : le couple de règles de production (b) n'est pas équivalent aux quatre règles (a).

La règle est alors la suivante : seule la porte de sortie (c'est-à-dire la porte qui produit un net qui n'est pas fusionné) peut être non-combinatoire.

Soit un ensemble de règles de production R. Soient deux couples de règles de production $G_0 = G_0^+ \rightarrow x_0 \nearrow, G_0^- \rightarrow x_0 \searrow$ et $G_1 = G_1^+ \rightarrow x_1 \nearrow, G_1^- \rightarrow x_1 \searrow$.

Définition 5.1 *G_0 et G_1 sont **fusionnables** si et seulement si toutes les conditions suivantes sont vraies :*

- *G_1^+ ou G_1^- dépendent de x_0 (c'est-à-dire que $G_1^+(x_0 = 0) \neq G_1^+(x_0 = 1)$ ou $G_1^-(x_0 = 0) \neq G_1^-(x_0 = 1)$).*
- *$\forall G \in R - \{G_1\}$, G^+ et G^- ne dépendent pas de x_0.*
- *G_0 est combinatoire.*

Définition de la fusion sur un ensemble de règles de production

Formellement, soit un ensemble de règles de production R sur l'ensemble de variables V. Soit $x_0 \in V$ une variable que l'on veut fusionner.

Soient $G_0 = (G_0^+ \rightarrow x_0 \nearrow, G_0^- \rightarrow x_0 \searrow) \in R^2$ et $G_1 = (G_1^+ \rightarrow x_1 \nearrow, G_1^- \rightarrow x_1 \searrow) \in R^2$ deux couples de règles de production de R fusionnables.

Définition 5.2 *Le couple de règles de production issu de la* ***fusion*** *de G_0 et de G_1 est le couple de règles de production G_{01} suivant :*

$$\begin{aligned} {G_1^+}_{x_0=G_0^+} &\rightarrow x_1 \nearrow \\ {G_1^-}_{x_0=G_0^+} &\rightarrow x_1 \searrow \end{aligned}$$

L'ensemble de règles de production issu de la ***fusion*** *de G_0 et de G_1 est l'ensemble $R - \{G_0, G_1\} \cup \{G_{01}\}$.*

Puisque G_0 est combinatoire, on a $\overline{G_0^+} = G_0^-$, donc toute l'information est contenue dans G_0^+. Remplacer x_0 par G_0^+ revient donc à remplacer $\overline{x_0}$ par G_0^- ; les règles de production obtenues après fusion ont donc le même comportement que les originales.

Preuve

On souhaite démontrer le théorème suivant :

Théorème 5.1 *Si R' est un ensemble de règles de production issu de la fusion de deux règles de production de R, et si R est quasi insensible aux délais, alors R' est quasi insensible aux délais.*

On note G_0 et G_1 les couples de règles fusionnés, et G_{01} le résultat de cette fusion. Supposons que R est quasi insensible aux délais. Alors, il est stable et non-interférant. On montre tout d'abord que R' est non-interférant. Il suffit de montrer que G_{01} est non-interférant, puisque toutes les autres règles sont identiques.

Pour cela, il faut montrer que ${G_1^+}_{|x_0=G_0^+} \wedge {G_1^-}_{|x_0=G_0^+}$ est toujours faux. Or, G_1 est non-interférant, puisque $G_1 \in R$. Donc $G_1^+ \wedge G_1^-$ est toujours faux. Ceci est en particulier vrai pour $x_0 = G_0^+$. Donc G_{01} est non-interférant, et par conséquent R' est non-interférant.

On montre ensuite que R' est stable. Il suffit de montrer que les deux règles de G_{01} sont stables. Considérons la règle ${G_1^+}_{|x_0=G_0^+} \rightarrow x_1 \nearrow$.

On veut montrer que ${G_1^+}_{|x_0=G_0^+}$ ne peut passer de vrai à faux avant que x_1 ne soit passé à 1. Or, G_1 est stable, donc G_1^+ ne peut passer de vrai à faux avant que x_1 ne soit passé à 1. Ceci est vrai en particulier pour $x_0 = G_0^+$. Donc la règle est stable.

On montre de même que ${G_1^-}_{|x_0=G_0^+} \rightarrow x_1 \searrow$ est stable, donc R' est stable.

L'ensemble de règles de production R' est stable et non-interférant, il est donc quasi insensible aux délais. On a donc montré le théorème.

6 Conclusion

Dans ce chapitre, on a démontré que la technique de synthèse développée au cours de cette thèse, et présentée dans le chapitre précédent, génère bien des circuits quasi insensibles aux délais.

Pour cela, la propriété de quasi insensibilité aux délais a été caractérisée formellement, à l'aide de l'article [8]. Cet article donne une condition nécessaire et suffisante pour qu'un ensemble de règles de production soit quasi insensible aux délais.

Puisque la propriété de quasi insensibilité aux délais est définie sur un ensemble de règles de production, il a fallu modéliser le circuit issu de notre méthode de synthèse par un ensemble de règles de production.

Dans un premier temps, on a démontré qu'un circuit issu de la synthèse DIMS est quasi insensible aux délais. Cette démonstration fait deux hypothèses sur le MDD. Tout d'abord, il doit être bien formé : on a vu au chapitre 3 que la structure de MDD permet d'écrire des MDD mal formés, qui décrivent un circuit ne pouvant être quasi insensible aux délais. Le MDD doit aussi être correctement acquitté : il doit générer un signal d'acquittement pour toutes les entrées qui ont participé à la production d'une sortie, à chaque cycle de calcul.

Dans un second temps, on a démontré qu'un circuit issu de la synthèse en portes deux entrées est quasi insensible aux délais. C'est ce résultat qui est important, et qui motive tout ce chapitre. La démonstration se base sur le résultat précédent sur la synthèse DIMS. En effet, la synthèse en portes deux entrées est équivalente à une synthèse DIMS sur un ensemble de MDDs de profondeur 2.

Pour finir, on a démontré que la fusion de portes dans une netlist préserve la propriété de quasi insensibilité aux délais. Ce résultat permet d'effectuer une projection technologique du circuit issu de la synthèse en portes deux entrées : puisque le circuit est entièrement décomposé, la projection technologique n'aura à effectuer que des fusions de portes, et préservera donc la propriété de quasi insensibilité aux délais du circuit. L'algorithme de projection technologique fait l'objet d'un travail indépendant de cette thèse [45, 46].

Chapitre 6

Implémentation et Résultats

1 Présentation de TAST

TAST est une collection d'outils destinés à la synthèse de circuits asynchrones, développée au laboratoire TIMA. La version de TAST présentée ici est la version développée dans cette thèse.

La figure 6.1 présente le flot de conception de TAST.

TAST s'articule autour d'un format intermédiaire, qui est basé sur les réseaux de Pétri. Il se compose de plusieurs modules indépendants. Le module front-end est chargé de transformer un programme CHP, qui est notre format d'entrée, pour générer la forme intermédiaire du réseau de Pétri.

Les modules de vérification et simulation s'articule autour de ce format intermédiaire en réseau de pétri, ainsi que le back-end qui est chargé, à partir de ce format intermédiaire, de générer une forme structurelle du circuit : une netlist hiérarchique de portes.

Le module qui nous intéresse ici est le back-end : c'est dans ce module que s'intègre le processus de synthèse.

La première étape, à partir du format intermédiaire en réseau de Pétri, est de générer le MDD modélisant le circuit que l'on souhait synthétiser, puisque la méthode de synthèse repose sur ce modèle. Cependant, comme on l'a vu au chapitre 4, cette étape n'est pas encore finalisée ; elle fait l'objet d'un travail à part entière.

Sur le MDD, on peut lancer les algorithmes d'optimisation présentés chapitre 4, qui permettent de réduire la taille du MDD. Puis le MDD peut être synthétisé en une netlist de portes, à l'aide de l'algorithme de synthèse en portes à deux entrées. Cette netlist est quasi insensible aux délais, et elle est totalement décomposée en portes à deux entrées. Elle ne cible pas de bibliothèque particulière : ses portes instancient des cellules génériques (ou logique, et logique, inverseur, porte de Muller).

L'étape de projection technologique, qui fait l'objet d'un travail de recherche à part [45, 47], permet de cibler une bibliothèque de cellules : la netlist de cellules génériques est transformée, afin d'instancier les cellules complexes de la bibliothèque cible, et ainsi optimiser cette netlist. L'optimisation peut se faire selon divers critères (surface, vitesse, consommation), puisque les caractéristiques des cellules de la bibliothèque sont connues. La transformation effectue uniquement des fusions de portes, afin de préserver la quasi insensibilité aux délais du circuit. Ceci est possible puisque la netlist générique ne comporte que des portes à deux entrées : elle est

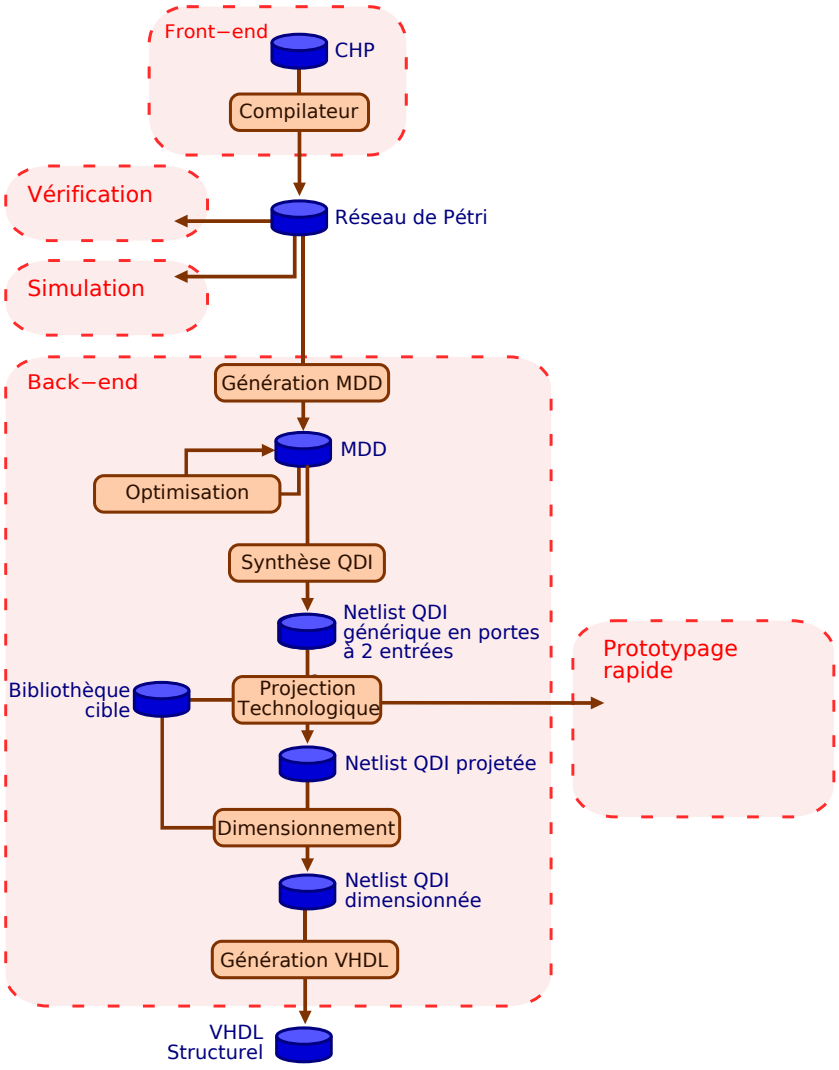


FIG. 6.1 – le flot de conception de TAST.

totalement décomposée.

Une fois la bibliothèque ciblée, il est encore possible d'optimiser le circuit. En effet, chaque cellule de la bibliothèque est disponible en plusieurs versions, de sortance différentes. Le dimensionnement consiste à choisir, pour chaque instance de la netlist, la taille de la cellule, afin d'optimiser la netlist selon le critère que l'on souhaite (surface, vitesse ou consommation). Cette optimisation se fait en tenant compte d'un modèle physique qui prend en compte le comportement analogique des portes. Cette étape a fait l'objet d'un travail à part entière, qui n'a malheureusement pas été publié. La seule transformation effectuée sur la netlist est ici l'annotation de chaque instance, afin de mémoriser quel *drive* de la cellule il faut utiliser ; la netlist reste donc QDI après cette étape.

La dernière étape du flot de synthèse écrit la netlist obtenue dans un fichier en VHDL.

Ce fichier VHDL structurel peut ensuite être utilisé dans un outil standard de placement/routage : le fait que le circuit ne soit pas synchrone mais quasi insensible aux délais n'a plus que très peu d'incidence sur la suite. Techniquement, il faut

vérifier que l'hypothèse des fourches isochrones est bien vérifiée ; en pratique cette hypothèse est très peu contraignante, et ne pose pas de problème.

L'outil développé au cours de cette thèse implémente ces fonctionnalités ; il est codé en langage C++ et comporte environ 30.000 lignes de code, qui forment deux bibliothèques ainsi qu'un programme d'interface utilisateur (invite de commande).

2 Circuits d'évaluation

Puisque l'outil fonctionne, on valider la méthode de synthèse. Pour cela, on a besoin de circuits de test.

Cependant, comme on l'a vu dans le chapitre 4, la génération de MDD à partir de code CHP n'est pas totalement automatisée. Or, on souhaite générer des circuits d'une taille conséquente, qui permette de valider la méthode de manière raisonnable. Écrire les MDDs à la main n'est donc pas une alternative viable : il n'est pas raisonnable de dépasser une centaine de nœuds dans de telles conditions.

L'idée retenue pour générer les MDDs de test consiste à trouver un circuit suffisamment régulier pour qu'il soit facile d'écrire un programme simple automatisant la construction du MDD : ainsi, il est faisable de générer directement des MDDs de taille importante.

Les circuits implémentant des opérations arithmétiques sont très réguliers, et bien documentés : additionneur, multiplieur, diviseur. C'est ce type de circuits qui va nous servir pour évaluer la méthode de synthèse.

2.1 Principe

Les circuits que l'on veut synthétiser sont réguliers, c'est-à-dire qu'ils sont composés de quelques blocs de base simples, instanciés un grand nombre de fois.

Le principe de la génération du MDD est le suivant :

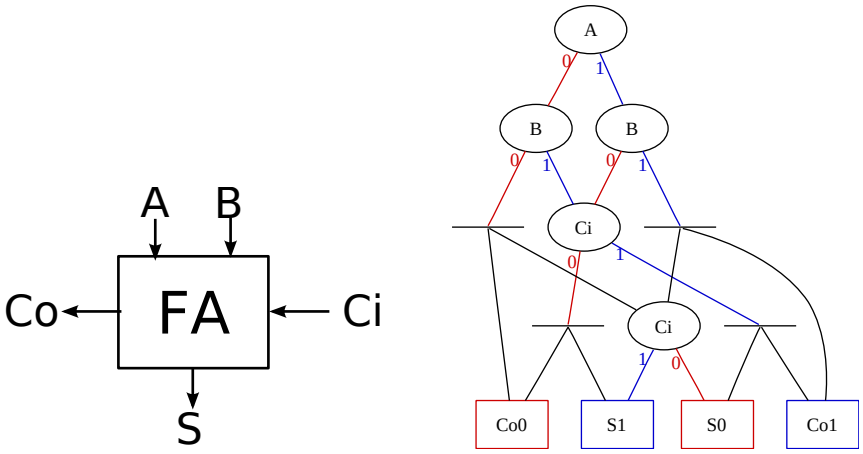
- Chaque bloc de base est écrit à la main sous forme de MDD.
- Dans le programme, à chaque bloc de base correspond une fonction. Cette fonction prend en paramètre les variables d'entrée et sortie du bloc que l'on souhaite instancier, et construit le MDD, en utilisant ces variables pour étiqueter les nœuds du MDD.
- Pour chaque bloc de base instancié dans le circuit, la fonction correspondante est appelée. Des variables internes sont utilisées pour les nets reliant les blocs de base, lorsque c'est nécessaire.
- Le MDD obtenu est la réunion de tous les MDDs. C'est un MDD composé : il utilise des variables internes pour connecter les différentes composantes du MDD.

Cette technique ne permet de générer que le chemin direct. Cependant, le chemin d'acquittement est très facile à générer : puisqu'à chaque cycle d'exécution, on consomme toutes les entrées, et on émet une valeur sur toutes les sorties il suffit d'un chemin du MDD regroupant les acquittements de toutes les sorties, qui produit les acquittements de toutes les entrées. Le protocole implémenté ainsi est le protocole séquentiel. Pour obtenir un circuit implémentant le protocole WCHB, il suffit d'ajouter un half-buffer en sortie du circuit.

2.2 Composants de base

On commence par présenter les deux composants de base utilisés pour implémenter les opérateurs logiques : le full adder et le soustracteur conditionnel

Le *Full adder*



(a) porte (b) MDD

FIG. 6.2 – le bloc de base *Full Adder*.

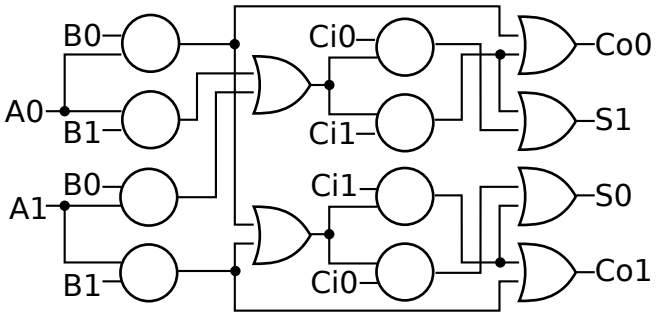


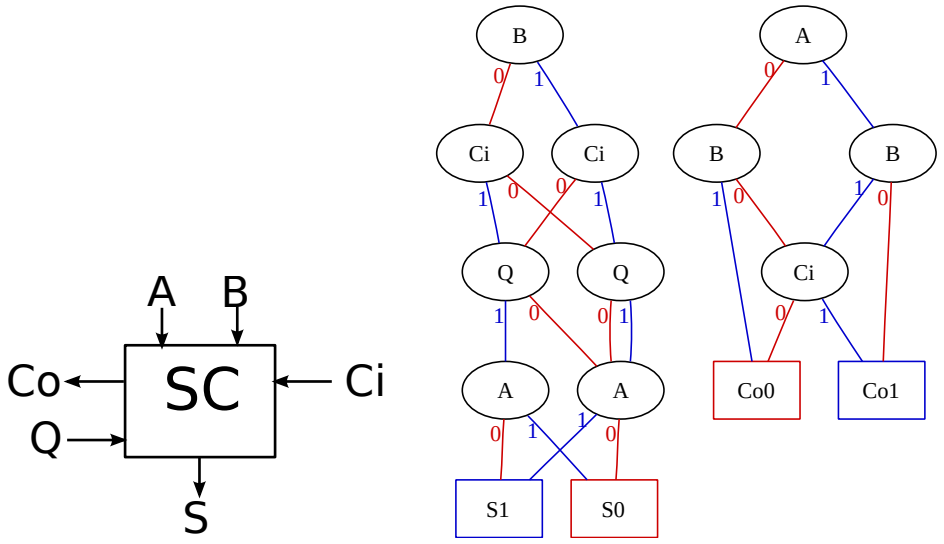
FIG. 6.3 – Circuit implémentant le *Full Adder*.

Le full adder a 3 entrées : A, B et la retenue entrante Ci. Il fait la somme des 3 entrées, et la propage sur les 2 bits de sortie Co et S (puisque le résultat est compris entre 0 et 3), le bit de poids fort étant la retenue sortante Co. Les figures 6.2 et 6.3 montrent le MDD et le circuit implémentant un full adder.

On peut remarquer que la retenue sortante Co ne dépend pas de la retenue entrante Ci dans les cas où ce n'est pas nécessaire : si A et B sont à 0, alors Co vaudra forcément 0, indépendemment de Ci ; de même si A et B sont à 1, Co vaudra forcément 1, indépendemment de Ci. Cette remarque est importante, puisqu'elle permet, en moyenne, de couper la chaîne de dépendance de la sortie.

Le Soustracteur Conditionnel

Le soustracteur conditionnel a 4 entrées : A, B, la retenue entrante Ci et la condition Q, et deux sorties, S et la retenue sortante Co. La retenue sortante se



(a) porte (b) MDD

FIG. 6.4 – le bloc de base *Soustracteur Conditionnel. La retenue sortante Co ne dépend jamais de la condition Q.*

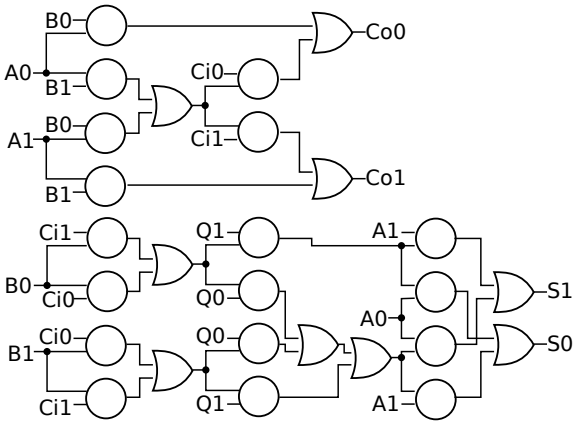


FIG. 6.5 – Circuit implémentant le *Soustracteur Conditionnel.*

comporte comme dans le full adder, l'entrée B étant inversée (il effectue donc une soustraction et non une addition). Elle est indépendante de Q, et ne dépend de Ci que dans les cas où c'est nécessaire. Si $Q = 1$, la sortie S se comporte elle aussi comme dans le full adder, l'entrée B étant inversée. Si $Q = 0$, la valeur de A est propagée sur la sortie S.

Ainsi, Q sélectionne si l'on effectue la soustraction, ou si l'on propage la valeur de A. Les figures 6.4 et 6.5 montrent le MDD et le circuit implémentant un soustracteur conditionnel.

Le fait que Co ne dépende pas de Q est indispensable pour le bon fonctionnement des circuits : Q sera calculé à partir de Co, et on aurait donc une boucle de dépendance.

2.3 Présentation des circuits

On détaille la structure de chacun des opérateurs arithmétiques considérés. Dans chaque cas, l'on choisit la structure la plus simple : le but n'est pas de générer l'opérateur le plus rapide, mais d'obtenir des circuits sur lesquels on puisse évaluer la synthèse développée dans cette thèse.

L'additionneur

La figure 6.6 présente la structure de l'additionneur.

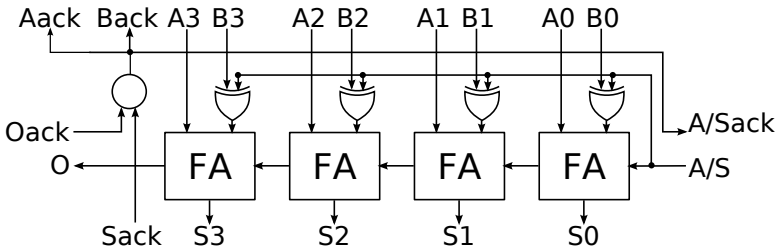


FIG. 6.6 – Structure de l'additionneur 4 bits.

C'est la structure d'additionneur la plus simple, elle est linéaire. L'entrée A/S permet de sélectionner si l'on souhaite additionner ou soustraire les deux nombres : pour soustraire, la retenue entrante passe à 1, et on inverse chaque bit de l'entrée B.

Cette structure est très lente pour un additionneur synchrone, puisque le chemin critique suit le chemin des retenues, qui passe par chaque full adder : le délai du chemin critique est proportionnel au nombre de bits.

Cependant, pour un circuit asynchrone, le problème ne se pose pas. En effet, ce n'est pas le pire cas qui nous intéresse, mais le cas moyen, puisque le délai du circuit s'adapte aux valeurs des entrées. Or, étant donné que la retenue sortante du bloc full adder ne dépend pas toujours de la retenue entrante, le chemin de retenues ne traverse tous les blocs *full adder* que pour de rares cas : en moyenne, le délai de fonctionnement du circuit est beaucoup plus court (logarithme du nombre de bits).

Le multiplieur

La figure 6.7 présente la structure du multiplieur.

Cette structure calcule le produit partiel de chaque couple de bits des entrées, puis regrouppe les produits partiels de même poids avec des full adders, selon la formule suivante :

$$\begin{aligned}\sum_{k=0}^{2n-2} s_k 2^k &= \left(\sum_{i=0}^{n-1} a_i 2^i\right)\left(\sum_{j=0}^{n-1} b_j 2^j\right) \\ &= \sum_{j=0}^{n-1}\sum_{i=0}^{n-1} a_i b_j 2^{i+j} \\ &= \sum_{k=0}^{2n-2}\left(\sum_{i=0}^{k} a_i b_{k-i}\right) 2^k\end{aligned}$$

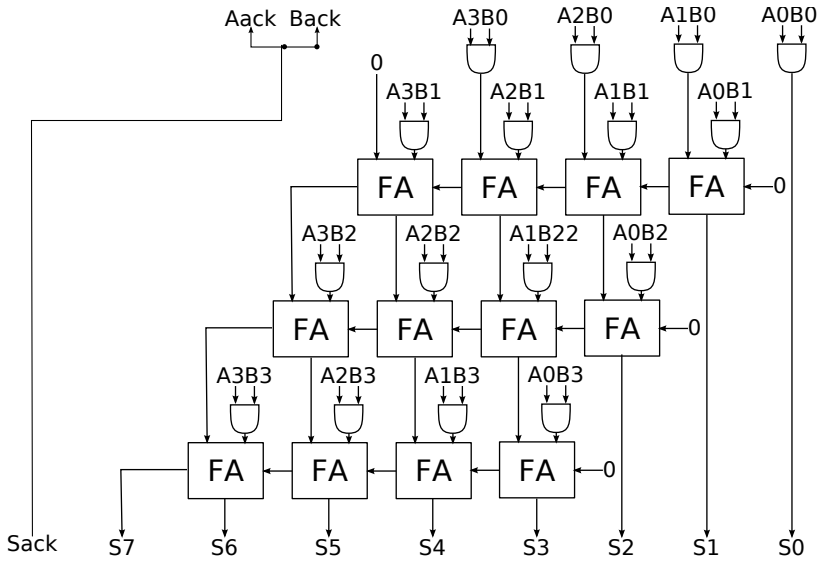


FIG. 6.7 – Structure du multiplieur 4 bits.

Chaque colonne du circuit calcule l'un des s_k, qui est le bit de poids k de la somme. Les retenues sont propagées vers le calcul du bit de poids supérieur. Il est à noter que l'algorithme présenté ici est le même que celui que l'on apprend à l'école primaire (à ceci près qu'il est en base 2).

La structure est quadratique : il y a $n(n-1)$ full adders, n étant le nombre de bits (A et B sont sur n bits, S sur $2n$ bits).

Diviseur

La figure 6.7 présente la structure du diviseur.

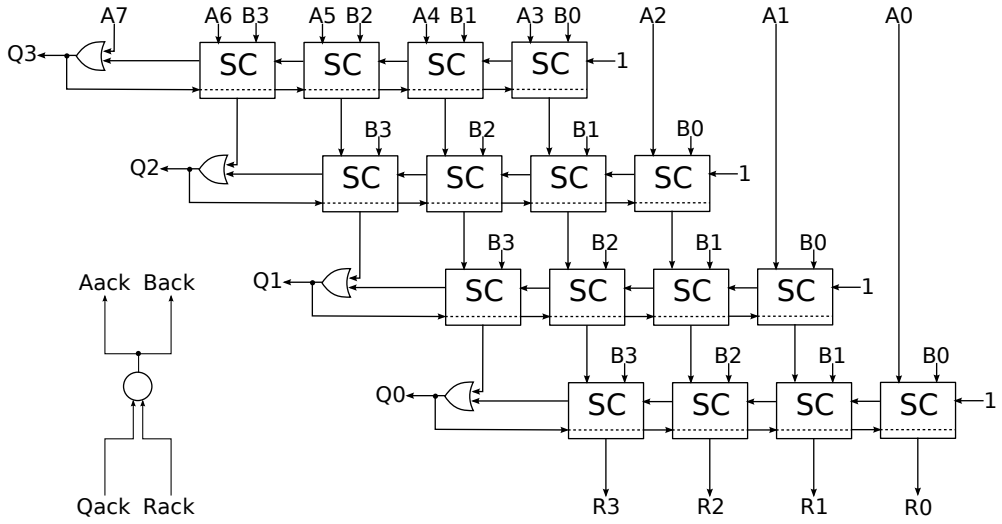


FIG. 6.8 – Structure du diviseur 4 bits.

Cette structure implémente l'algorithme de division euclidienne : le bit de poids fort du quotient est obtenu en comparant le numérateur au dénominateur, décalé de $n-1$ bits ; s'il est à 1, on soustrait le dénominateur décalé au numérateur, et on recommence pour obtenir le bit de poids suivant.

Comme pour la multiplication, cette structure est quadratique. Elle utilise le bloc soustracteur conditionnel ; chaque ligne de soustracteurs conditionnels implémente une étape de l'algorithme de division euclidienne, et calcule donc l'un des bits du quotient.

3 Résultats

La figure 6.9 présente la taille des circuits obtenus. La grandeur mesurée est la surface, c'est-à-dire en fait la somme des surfaces des cellules utilisées (la véritable surface du circuit ne pouvant être connue avant le placement-routage).

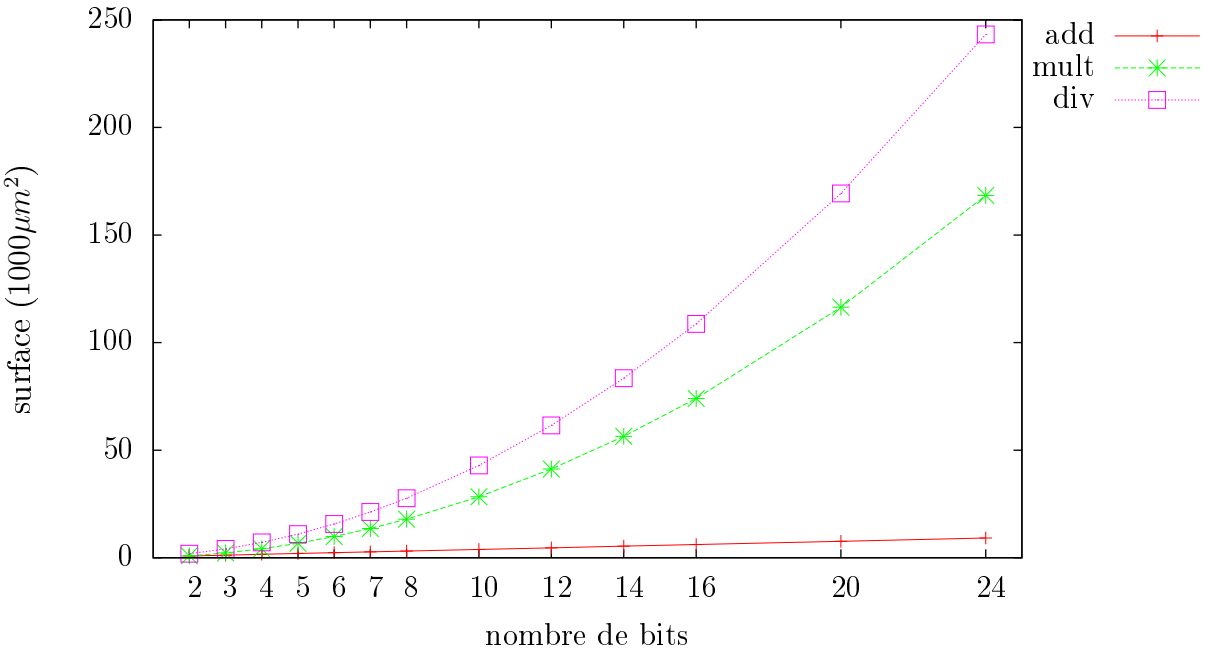


FIG. 6.9 – Taille des circuits obtenus, en fonction du nombre de bits

Il est à noter que ces résultats sont bruts, c'est-à-dire que le circuit évalué est le circuit en portes deux entrées directement généré par la synthèse. En particulier, aucune projection technologique intelligente n'a pas été effectuée : les portes à deux entrées ont directement été mises en correspondance avec une cellule de la bibliothèque cible, sans effectuer aucune fusion, et donc sans utiliser les cellules de la bibliothèque à plus de deux entrées. La projection technologique fait l'objet d'un travail à part entière, qui n'est pas tout à fait intégré à l'environnement TAST à l'heure de l'écriture de ce document.

4 Conclusion

J'ai implémenté la méthode de synthèse développée dans les chapitres précédents, sous la forme d'un outil de synthèse intégré dans TAST. Cet outil de synthèse prend en entrée un fichier basé sur XML, qui spécifie le MDD que l'on souhaite synthétiser.

Le MDD est chargé en mémoire, optimisé, puis synthétisé. La netlist obtenue est exportée au format VHDL. L'outil de synthèse a été testé, il fonctionne, même sur des circuits de plusieurs milliers de portes.

De plus, j'ai mis en place un petit ensemble de circuits d'évaluations, qui permettent de valider la méthode de synthèse. Les MDDs sont générés automatiquement par un programme indépendant ; on peut choisir le nombre de bits sur lesquels on travaille, ce qui permet de générer des circuits plus ou moins gros. Ces circuits implémentent les opérations arithmétiques élémentaires.

Conclusion

Ce travail de thèse comporte trois composantes :
- une composante appliquée, dans laquelle j'ai défini le modèle de MDD, et la méthode de synthèse automatique de circuits quasi insensibles aux délais à partir de ce modèle,
- une composante théorique, dans laquelle j'ai étudié la modélisation de circuits quasi insensibles aux délais, et démontré que la synthèse produisait des circuits qui sont quasi insensibles aux délais,
- et une composante développement, dans laquelle j'ai développé et testé la méthode de synthèse.

Le but de cette thèse était de contribuer à développer un outil permettant de synthétiser automatiquement des circuits, en garantissant que les circuits synthétisés sont quasi insensibles aux délais.

La propriété de quasi insensibilité aux délais d'un circuit est une propriété complexe, qu'il est très difficile de vérifier : déterminer si un circuit est quasi insensible aux délais ou non est un problème à part entière, qu'on ne sait généralement pas résoudre simplement. L'approche choisie dans cette thèse consiste à contourner le problème : plutôt que de vérifier que le circuit synthétisé est bien quasi insensible aux délais, on s'assure que sous certaines conditions sur le modèle, le flot de synthèse génère forcément un circuit QDI. Cette approche a été prouvée formellement.

Cependant, les possibles transformations du circuit après la synthèse peuvent compromettre la quasi insensibilité aux délais du circuit. Il faut donc s'assurer que les transformations effectuées ne peuvent pas influer sur cette propriété : c'est la raison pour laquelle nous n'autorisons que la fusion de portes.

Pourtant, il est souhaitable d'effectuer des transformations complexes sur le circuit, afin de l'optimiser. Pour cela, on a défini un modèle à base de MDD. Ce modèle permet d'effectuer des optimisations avant synthèse. Ce modèle est un graphe de décision qui permet de décrire le circuit. Contrairement au modèle de BDD, très utilisé dans le monde synchrone, le MDD ne se limite pas à la définition de la fonction booléenne des sorties en fonction des entrées. En effet, cette définition ne serait pas suffisante pour définir le comportement d'un circuit asynchrone, qui doit déterminer quelles entrées sont consommées et doivent être acquittées.

Sur ce modèle de MDD, divers algorithmes d'optimisation ont été développés. Ils permettent, en réduisant la taille du MDD, de réduire la taille du circuit qui sera synthétisé à partir du MDD, sans que cela compromette la quasi insensibilité aux délais du circuit. Puisque le MDD optimisé est toujours un MDD, le circuit synthétisé à partir du MDD optimisé est quasi insensible aux délais. Il suffit de vérifier que le MDD optimisé vérifie toujours les contraintes sur lesquelles repose la preuve de quasi insensibilité aux délais.

Cette méthode est novatrice : les autres méthodes de synthèse existantes n'intègrent aucune vérification de la quasi insensibilité aux délais des circuits générés, ni théorique, ni sous forme de code.

La synthèse est effectuée en portes deux entrées. Ainsi, le circuit est totalement décomposé, et l'on peut effectuer une projection technologique sur une bibliothèque cible en n'effectuant que des fusions de portes (qui est la seule transformation autorisée, afin de préserver la quasi insensibilité aux délais du circuit).

Concernant la validation de la méthode, un ensemble de circuits de tests a été mis en place. Les circuits de test utilisés sont des opérateurs arithmétiques. Ces circuits sont très réguliers, ce qui permet de facilement générer des circuits de tailles diverses, en faisant simplement varier le nombre de bits.

Perspectives

Plusieurs pistes peuvent être considérées pour continuer ce travail.

Tout d'abord, il faut finir l'automatisation de l'étape de génération de MDD à partir d'un programme CHP. Cette étape est intimement liée au travail réalisé au cours de cette thèse. Le point difficile dans cette étape est le calcul des acquittements, qui n'est pas spécifié directement dans le langage CHP : il peut y avoir plusieurs manières d'acquitter un même programme CHP, ce qui amène à faire des choix, ce qui n'est pas toujours facile à faire automatiquement.

À plus long terme, peut-être serait-il souhaitable de créer un nouveau langage de spécification de circuits asynchrones, qui serait adapté, et aiderait le travail de synthèse. Ce langage serait fonctionnel, afin d'intégrer un maximum de parallélisme. En effet, dans un langage impératif comme CHP, la séquentialité des instruction n'est pas toujours synthétisable, ce qui pose des problèmes. Dans un langage fonctionnel, les seules séquentialités seraient les dépendances de données, qui ne posent aucune problème. De plus, ce langage devra spécifier explicitement l'acquittement. Ainsi, c'est le programmeur qui devra choisir comment calculer les acquittements. Un programme dans ce langage pourrait être généré automatiquement à partir d'un langage plus haut niveau, si le programmeur ne souhaite pas se soucier des acquittements. Une telle structure permettrait d'isoler le problème de calcul d'acquittements de la synthèse.

Concernant l'optimisation des circuits, une approche qui n'a pas été envisagée concerne le codage des données. On a vu que le codage des données en 1 parmi n est très coûteux, au final, sur la taille du circuit, qui est le principal handicap des circuits quasi insensibles aux délais face aux circuits synchrones. Or, il existe d'autres codes insensibles aux délais que le 1 parmi n, qui sont plus compacts. Cette piste ne nécessiterait pas forcément de modifications au niveau du MDD : en effet, dans le MDD, chaque nœud non-terminal spécifie quel nœud activer en fonction de la valeur lue sur la variable, mais sans spécifier comment cette valeur est codée. Par contre, le principe d'équivalence entre un arc du MDD et un net du circuit ne serait pas conservé. L'algorithme de synthèse, ainsi que la preuve, devraient vraisemblablement êtres réécrits en tenant compte de ceci.

Bibliographie

[1] Kam (T.), Villa (T.), Brayton (R.) et Sangiovanni-Vincentelli (A.), « Multi-valued decision diagrams : theory and applications », *Multiple-Valued Logic*, vol. 4, n° 1-2, 1998, p. 9–62.

[2] Babu (H.), Md (H.) et Sasao (T.), « Heuristics to minimize multiple-valued decision diagrams », *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 2000, p. 2498–2504.

[3] Dinh-Duc (A.-V.), *Synthèse Automatique de Circuits Asynchrones QDI*. Thèse de doctorat, INP Grenoble, march 2003.

[4] Udding (J.), « A formal model for defining and classifying delay-insensitive circuits and systems », *Distributed Computing*, vol. 1, n° 4, 1986, p. 197–204.

[5] Martin (A. J.). « The limitations to delay-insensitivity in asynchronous circuits. sixth mit conference on advanced research in vlsi, ed. wj dally », 1990.

[6] van Berkel (K.), « Beware the isochronic fork », *Integration, the VLSI Journal*, vol. 13, n° 2, 1992, p. 103–128.

[7] Sretasereekul (N.) et Nanya (T.), « Eliminating isochronic-fork constraints in quasi-delay-insensitivecircuits », *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, 2001, p. 437–442.

[8] Manohar (R.) et Martin (A. J.). « Quasi-delay-insensitive circuits are turing-complete », janvier 1996.

[9] Martin (A. J.), « Programming in vlsi : From communicating processes to delay-insensitive circuits ». Rapport technique, Pasadena, CA, USA, 1989.

[10] Kung (D.), « Hazard-non-increasing gate-level optimization algorithms », *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, 1992, p. 631–634.

[11] Verhoeff (T.), « Delay-insensitive codes–an overview », *Distributed Computing*, vol. 3, n° 1, 1988, p. 1–8.

[12] Worm (F.), Thiran (P.) et Ienne (P.), « A unified coding framework for delay-insensitivity », *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, 2005, p. 201–211.

[13] Kondratyev (A.), Neukom (L.), Roig (O.) *et al.*, « Checking delay-insensitivity : 10^4 gates and beyond », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2002, p. 149–157.

[14] Yahya (E.) et Renaudin (M.), « Qdi latches characteristics and asynchronous linear-pipeline performance analysis. », dans *PATMOS*, p. 583–592, 2006.

[15] Peeters (A.) et Solutions (H.), « Bringing handshake technology to the open market », *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, 2004.

[16] Hoare (C.), « Communicating sequential processes », *Communications of the ACM*, vol. 21, n° 8, 1978, p. 666–677.

[17] Bardsley (A.) et Edwards (D.), « Compiling the language Balsa to delay-insensitive hardware », *Hardware Description Languages and their Applications (CHDL)*, 1997, p. 89–91.

[18] Edwards (D.) et Bardsley (A.), « Balsa : An Asynchronous Hardware Synthesis Language », *The Computer Journal*, vol. 45, n° 1, 2002, p. 12.

[19] Wong (C.) et Martin (A.), « High-level synthesis of asynchronous systems by data-driven decomposition », *Design Automation Conference, 2003. Proceedings*, 2003, p. 508–513.

[20] Teifel (J.) et Manohar (R.), « Static tokens : using dataflow to automate concurrent pipeline synthesis », *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, 2004, p. 17–27.

[21] Martin (A. J.), « Programming in vlsi : From communicating processes to delay-insensitive circuits », dans Hoare (C. A. R.), éditeur, *Developments in Concurrency and Communication*, p. 1–64. Addison-Wesley, Reading, MA, 1990.

[22] Duc (A.), Fesquet (L.) et Renaudin (M.), « Synthesis of QDI Asynchronous Circuits from DTL-style Petri-Net », *IWLS-02, 11th IEEE/ACM Internat. Workshop on Logic & Synthesis, New Orleans, Louisiana, June*, 2002, p. 4–7.

[23] Cortadella (J.), Kishinevsky (M.), Kondratyev (A.) *et al.*, « Petrify : A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers », *IEICE Transactions on Information and Systems*, vol. 80, n° 3, 1997, p. 315–325.

[24] Fant (K.) et Brandt (S.), « NULL Convention Logic », *Theseus Logic : Setting the Standard for Clockless Systems*, 1997.

[25] Ligthart (M.), Fant (K.), Smith (R.) *et al.*, « Asynchronous design using commercial HDL synthesis tools », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000, p. 114–125.

[26] Smith (S.), DeMara (R.), Yuan (J.) *et al.*, « Optimization of NULL convention self-timed circuits », *Integration, The VLSI Journal*, vol. 37, n° 3, 2004, p. 135–165.

[27] Smirnov (A.), Taubin (A.), Karpovsky (M.) et Rozenblyum (L.), « Gate Transfer Level Synthesis as an Automated Approach to Fine-Grain Pipelining », *Workshop on Token Based Computing (ToBaCo)*, 2004.

[28] Manohar (R.), Lee (T.) et Martin (A.), « Projection : A Synthesis Technique for Concurrent Systems », *Computer*, vol. 256, 1999, p. 80.

[29] Yoneda (T.), Onda (H.) et Myers (C.), « Synthesis of speed independent circuits based on decomposition », *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, 2004, p. 135–145.

[30] Wong (C.) et Martin (A.), « Data-driven Process Decomposition For the Synthesis of Asynchronous Circuits », *Proc. IEEE Conference on Electronic Circuits and Systems*, 2001.

[31] Scholl (C.) et Molitor (P.), « Efficient ROBDD based computation of common decomposition functions of multi-output boolean functions », *Novel approaches in logic and architecture synthesis*, 1995, p. 57–63.

[32] Scholl (C.), Moller (D.), Molitor (P.) et Drechsler (R.), « BDD minimization using symmetries », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, n° 2, 1999, p. 81–100.

[33] Stanion (T.) et Sechen (C.), « Boolean division and factorization using binary decision diagrams », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, n° 9, 1994, p. 1179–1184.

[34] Lai (Y.), Pan (K.) et Pedram (M.), « OBDD-based function decomposition : algorithms and implementation », *IEEE Trans. Computer-Aided Design*, vol. 15, 1996, p. 977–990.

[35] Yun (K.), Lin (B.), Dill (D.) et Devadas (S.), « BDD-based synthesis of extended burst-mode controllers », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, n° 9, 1998, p. 782–792.

[36] Lin (B.), Devadas (S.) et IMEC (L.), « Synthesis of hazard-free multilevel logic under multiple-inputchanges from binary decision diagrams », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, n° 8, 1995, p. 974–985.

[37] Nielsen (C.), « Evaluation of Function Blocks for Asynchronous Design », *Proceedings of ACM, pp*, 1994, p. 454–459.

[38] Lemberski (I.) et Josephs (M.), « Optimal Two-Level Delay-Insensitive Implementation of Logic Functions », *Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, 2002, p. 92–100.

[39] David (I.), Ginosar (R.) et Yoeli (M.), « An efficient implementation of Boolean functions as self-timedcircuits », *Computers, IEEE Transactions on*, vol. 41, n° 1, 1992, p. 2–11.

[40] Cong (J.) et Ding (Y.), « FlowMap : an optimal technology mapping algorithm for delayoptimization in lookup-table based FPGA designs », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, n° 1, 1994, p. 1–12.

[41] Teslenko (M.) et Dubrova (E.), « Hermes : LUT FPGA technology mapping algorithm for area minimization with optimum depth », *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, p. 748–751.

[42] Burns (S.), « General conditions for the decomposition of state holding elements », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996, p. 48–57.

[43] Huffman (D.), « A method for the construction of minimum-redundancy codes », dans *Proceedings of the I.R.E.*, p. 1098–1102, sept 1952.

[44] Brégier (V.), Folco (B.), Fesquet (L.) et Renaudin (M.), « Modeling and synthesis of multi-rail multi-protocol QDI circuits », dans *Thirteenth International Workshop on Logic and Synthesis*, Temecula, California, juin 2004.

[45] Folco (B.), Brégier (V.), Fesquet (L.) et Renaudin (M.), « Synthesis of area optimized quasi delay insensitive circuits », dans *System On Chip 2005*, coll. « IFIP International Conference on Very Large Scale Integration », Perth, Australia, octobre 2005.

[46] Folco (B.), *(À paraître) Contribution à la synthèse automatique de circuits asynchrones QDI : application aux systèmes sécurisés.* Thèse de doctorat, Laboratoire TIMA, INP Grenoble, octobre 2007.

[47] Folco (B.), Brégier (V.), Fesquet (L.) et Renaudin (M.), *Technology Mapping for Area Optimized Quasi Delay Insensitive Circuits*, vol. 200 (coll. *IFIP International Federation for Information Processing Series*). Springer, 2005.

[48] Brzozowski (J.) et Ebergen (J.), « On the delay-sensitivity of gate networks », *IEEE Transactions on Computers*, vol. 41, n° 11, 1992, p. 1349–1360.

[49] Cook (J. N.), « Production rule verification for quasi-delay-insensitive circuits ». Rapport technique, juin 1993.

[50] van Berkel (K.), Huberts (F.) et Peeters (A.), « Stretching quasi delay insensitivity by means of extended isochronic forks », *Asynchronous Design Methodologies*, 1995, p. 99–106.

[51] Akella (V.), Vaidya (N. H.) et Redinbo (G. R.). « Limitations of VLSI implementation of delay-insensitive codes », janvier 1995.

[52] Nowick (S.) et Dill (D.), « Exact two-level minimization of hazard-free logic with multiple-input changes », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, n° 8, 1995, p. 986–997.

[53] chiung Cheng (F.). « Synthesizing iterative functions into delay-insensitive tree circuits », novembre 1997.

[54] Drechsler (R.) et Becker (B.), *Binary Decision Diagrams : Theory and Implementation.* Kluwer Academic Publishers, 1998.

[55] Cortadella (J.), Kishinevsky (M.), Burns (S.) *et al.*, « Lazy transition systems and asynchronous circuit synthesis withrelative timing assumptions », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, n° 2, 2002, p. 109–130.

[56] Sacker (M.), Brown (A.), Wilson (P.) et Rushton (A.), « A general purpose behavioural asynchronous synthesis system », *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, 2004, p. 125–134.

Résumé

Dans un circuit asynchrone, la synchronisation entre les blocs est locale : on s'affranchit ainsi des contraintes liées à l'horloge. Ces circuits sont plus robustes, modulaires, moins bruités, et ont une consommation dynamique plus faible que les circuits synchrones. Cependant, le manque d'outils de conception de tels circuits freine leur développement. Cette thèse a permi de développer une technique de synthèse automatique de circuits asynchrones quasi insensibles aux délais (QDI), qui sont particulièrement robustes. La méthode de synthèse permet de synthétiser un circuit totalement décomposé en portes logiques élémentaires, ce qui permet d'effectuer une projection technologique. De plus, une étude formelle réalisée durant la thèse démontre que les circuits synthétisés respectent la contrainte de quasi insensibilité aux délais. Cette technique de synthèse a été développé au sein du projet TAST. Elle a été validée sur un ensemble de circuits de tests.

Mot-clefs

synthèse, optimisation logique, asynchrone, quasi insensibilité aux délais (QDI), modélisation, diagrammes de décision multi-valués (MDD)

Title

Synthèse automatisée de circuits asynchrones optimisés prouvés Quasi Insensibles aux Délais

Abstract

In an asynchronous circuit, the synchronization between the blocs is local : the constraints due to the clock do not apply. These circuits are more robust, modular, have less noise and a lower dynamic power consumption that asynchronous circuits. However, the lack of design tools for such circuits prevents them from spreading widely. This thesis aimed at developping an automatic synthesis technique targeting asynchronous quasi delay insensitive (QDI) circuits, which are particularly robust. The technique synthesizes a circuit totally decomposed in elemetary logical gates, which allows a later technology mapping. Moreover, a formal study done during this thesis proves that the circuits synthesized respect the constraint of quasi delay insensitivity. This synthesis technique was developped in the TAST project. Is has been validated on a set of test circuits.

Keywords

synthesis, logical optimization, asynchronous, quasi delay insensitivity (QDI), modelization, multi-valued decision diagram (MDD)

Adrr : Laboratoire TIMA, 46 av. Félix Viallet, 38031 Grenoble

ISBN : 978-2-84813-105-4