UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

Conception d'un système de synthèse orienté-objet multiplateforme en vue d'une nouvelle méthode de synthèse

Mémoire de maîtrise
Spécialité : génie électrique

Dominic TARDIF

Jury : Frédéric MAILHOT (directeur)
Daniel DALLE
Philippe MABILLEAU

Sherbrooke (Québec) Canada Avril 2011

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-83670-5

Our file Notre référence
ISBN: 978-0-494-83670-5

RÉSUMÉ

La loi de Moore prédit que le nombre de composants dans un circuit double à tous les 18 mois. Cette augmentation permet de diminuer les délais dans ces composants, mais amènent une augmentation des délais liés aux interconnexions par rapport aux délais dans les composants et de la consommation de puissance.

Récemment, les délais dans les interconnexions sont devenus trop importants par rapport aux délais dans les portes logiques au point où la méthode de synthèse automatisée de circuits intégrés actuelle est devenue inadéquate. Puisque le traitement des interconnexions s'effectue lors de la synthèse physique, une nouvelle approche, inversant les étapes de la synthèse physique et de la synthèse logique, a été envisagée. La conception d'un système, utilisant un langage orienté-objet et offrant de la portabilité et une intégration de modules futurs, a été l'objet de cette recherche puisqu'un système utilisant un tel procédé n'a pas encore vu le jour.

Une plate-forme de synthèse a été développée et celle-ci a été testée à l'aide d'un module de gestion de budgets de délai. Premièrement, une lecture de la description logique du circuit provenant de la synthèse comportementale a été effectuée en utilisant un décomposeur analytique et un analyseur syntaxique. Ensuite, pendant cette lecture, un réseau booléen hiérarchique représentant le circuit a été bâti selon une infrastructure prédéfinie. Afin de pouvoir tester la plate-forme, des budgets de délais ont été assignés à chaque nœud du réseau en propageant temps d'arrivée et temps requis dans un circuit provenant d'une description logique hiérarchique complexe. Finalement, la gestion de budgets de délai a été faite par un algorithme conçu à cet effet et les résultats de celle-ci ont été analysés.

Le résultat obtenu est une plate-forme de synthèse capable de faire de la gestion de budget de délais sur les chemins critiques dans un circuit donné. De plus, celle-ci pourra être utilisée de nouveau pour d'autres projets liés à la synthèse de circuits. La pertinence de cette recherche repose sur la résolution d'un problème grandissant dans le monde de la synthèse automatisée des circuits intégrés.

Mots-clés : synthèse, budget, délai, réseau booléen, interconnexion

REMERCIEMENTS

Je tiens à remercier M. Frédéric Mailhot, mon directeur de recherche, de m'avoir accordé la chance de poursuivre mes études à la maîtrise et de m'avoir encouragé jusqu'à la fin. Sans lui, ce projet n'aurait pas eu lieu et j'en suis très reconnaissant.

Je tiens également à remercier MM. Philippe Mabilleau et Daniel Dalle, membres du comité, pour avoir accepté de réviser le présent travail.

Enfin, je tiendrais à remercier les quelques professeurs de la faculté que j'ai eu l'occasion de côtoyer tout au long de mes études supérieures et, plus spécialement, M. Ruben Gonzalez-Rubio pour m'avoir donné les outils essentiels au bon développement de ce projet.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ACRONYMES

Acronyme	Définition
ALU	Arithmetic Logic Unit
ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
CER	Critical Edge Reduction
CDFG	Control Data Flow Graph
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSM	Deep Submicron
EBNF	Extended Backus-Naur Form
GWT	Google Web Toolkit
ILP	Integer Linear Programming
LSI	Large Scale Integration
LSS	Logic Synthesis System
MIS	Multiple-level Interactive System
MISA	«Maximum Independent Set»-based Algorithm
MSI	Medium Scale Integration
MVC	Model-View-Controller
MWIS	Maximum Weighted Independent Set
NP	Nondeterministic Polynomial time
PLA	Programmable Logic Array
RTL	Register Transfer Level
SP	Series-Parallel
SSI	Small Scale Integration
SIS	Sequential Interactive System
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VIS	Verification Interacting with Synthesis
VLSI	Very Large Scale Integration
ZSA	Zero Slack Algorithm

CHAPITRE 1

INTRODUCTION

1.1 Historique

Les circuits intégrés, depuis leur apparition il y a plus de cinquante ans, n'ont cessé d'évoluer. Cependant, ceux-ci n'auraient probablement jamais vu le jour si ça n'avait été de la création du transistor en décembre 1947 dans les laboratoires Bell. Pendant l'été de 1958, Jack Kilby, un jeune ingénieur sortant à peine de l'université, arriva à mettre au point un circuit intégré regroupant quelques transistors sur un même substrat. À partir de cet instant, le monde de l'électronique a été en continuelle évolution.

1.1.1 Les débuts du circuit intégré

Les changements débutèrent par l'apparition de l'intégration à petite échelle (SSI pour «*Small Scale Integration*») où les circuits intégrés ne comportaient que quelques dizaines de transistors. Le prix des transistors, élevé au début des années 60, devint raisonnable quelques années plus tard en raison de la fabrication à grande échelle. Grâce à cela et au progrès dans la miniaturisation des composants, l'intégration à moyenne échelle (MSI pour «*Medium Scale Integration*») a vu le jour vers la fin des années 60. Les circuits intégrés, alors, étaient composés de quelques centaines de transistors et ils ne coûtaient pas vraiment plus chers que ceux de la première génération, mais ils permettaient plus d'avantages en termes de conception de circuits comme la possibilité d'avoir des systèmes plus complexes. Au milieu des années 70, l'intégration à grande échelle (LSI pour «*Large Scale Integration*»), pour des raisons économiques encore une fois, permettait d'avoir un circuit intégré comportant quelques dizaines de milliers de transistors à un coût relativement faible.

Dans les années 80, la taille des transistors avait tellement diminué et leur nombre augmenté qu'il était possible de concevoir un circuit intégré avec quelques centaines de milliers de transistors. Ce fut le début de l'intégration à très grande échelle (en anglais, «*Very Large Scale Integration*» ou VLSI). Déjà en 1971, le premier microprocesseur à voir le jour avait été l'Intel 4004, une unité centrale de traitement (CPU pour «*Central Processing Unit*») sur un seul circuit intégré. Grâce au VLSI, il était dorénavant possible de créer un microprocesseur plus complexe dans un espace plus petit.

1.1.2 Les effets de la loi de Moore

La figure 1.1 représente la loi de Moore en illustrant la relation entre la miniaturisation des transistors et leur quantité. En effet, Gordon Moore a remarqué, en 1965 [Intel, 2005], que la complexité des circuits avait tendance à doubler à tous les 2 ans environ et cela tient encore aujourd'hui, quoique nous assistions en ce moment à la fin de l'époque de la loi de Moore. Cette miniaturisation existe dans le but d'augmenter le rendement et la performance des circuits intégrés. En effet, le nombre d'opérations par seconde est directement proportionnel au nombre de transistors présents. La vitesse des circuits est aussi liée au degré d'intégration. La diminution constante de la taille des transistors a permis d'en augmenter le nombre pour un circuit de taille donnée. En plus du fait que la quantité de transistors dans un circuit intégré dépasse déjà le milliard, les circuits intégrés composés de transistors de 22 nm commencent à faire leur apparition et la technologie de 11 nm est prévue pour 2015 [Shilov, 2009]. Pour arriver à concevoir de tels circuits, nous avons dû avoir recours à la synthèse des circuits, et ce, depuis le milieu des années 70.

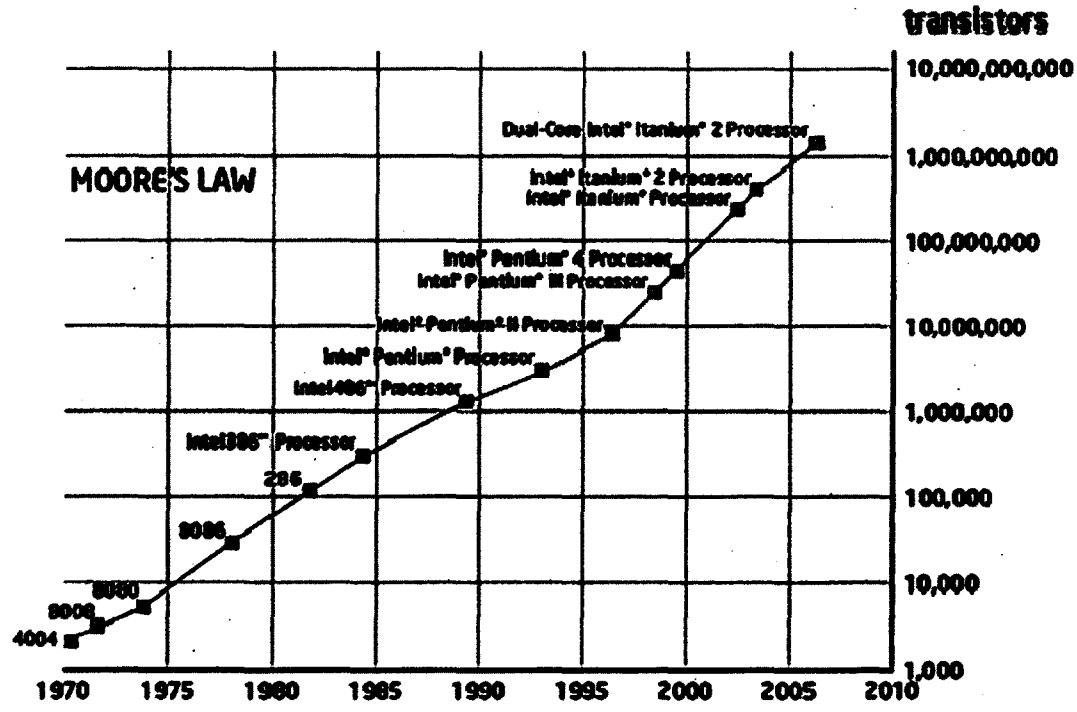


Figure 1.1 Évolution du nombre de transistors dans les processeurs suivant la loi de Moore depuis 1970 [Source : www.intel.com]

1.1.3 Les débuts de la synthèse des circuits

Celle-ci, elle aussi, s'est perfectionnée de façon parallèle au phénomène des circuits intégrés. Dans les années 70, la synthèse s'effectuait au moyen de diagrammes schématiques, ceux-ci étant composés de multiples portes logiques. Ensuite, au début des années 80, l'apparition des réseaux de logique programmable (en anglais, «*Programmable Logic Arrays*» ou PLA) correspondit au début des étapes de placement et de routage lors de la synthèse physique. Au milieu des années 80, la compagnie IBM sortit un système basé sur des règles, le LSS («*Logic Synthesis System*»).

Peu de temps après, l'université de Berkeley, en Californie, présenta un système basé sur des algorithmes, le MIS («*Multiple-level Interactive System*»), qui ne présente que l'optimisation de la logique combinatoire. Ce système a évolué vers le système SIS («*Sequential Interactive Synthesis*») qui y inclut l'optimisation de la logique séquentielle. En se basant sur le système SIS, l'outil VIS («*Verification Interaction with Synthesis*») permet d'intégrer la vérification, la simulation et la synthèse de systèmes à états finis [Brayton *et al.*, 1996]. Depuis la fin des années 80, la synthèse de haut niveau est comprise dans les systèmes automatisés de conception. Ce n'est que vers le milieu des années 90 que ceux-ci se sont vu incorporer la synthèse physique.

1.2 Identification de la problématique

Comme il a été vu précédemment, beaucoup de progrès ont été faits depuis les dernières années dans la synthèse des circuits. Cependant, de nouvelles problématiques se sont révélées en même temps que les circuits devenaient plus complexes. Pour bien identifier la problématique, un survol de la synthèse et un aperçu des inconvénients auxquels ont à faire face les ingénieurs d'aujourd'hui sont nécessaires.

1.2.1 La synthèse des circuits en 3 étapes

La conception (ou synthèse) des circuits est séparée en trois étapes distinctes. La synthèse de haut niveau permet de représenter le comportement du futur circuit sous la forme d'un ensemble d'opérations et de dépendances [De Micheli, 1994]. Les concepteurs de circuits numériques utilisent généralement des langages de description matérielle tels VHDL ou Verilog pour indiquer les fonctionnalités requises au système de synthèse de haut niveau. Le résultat de la synthèse de haut niveau est une description en termes de transfert de registre («*RTL*») et/ou d'interconnexion de portes logiques.

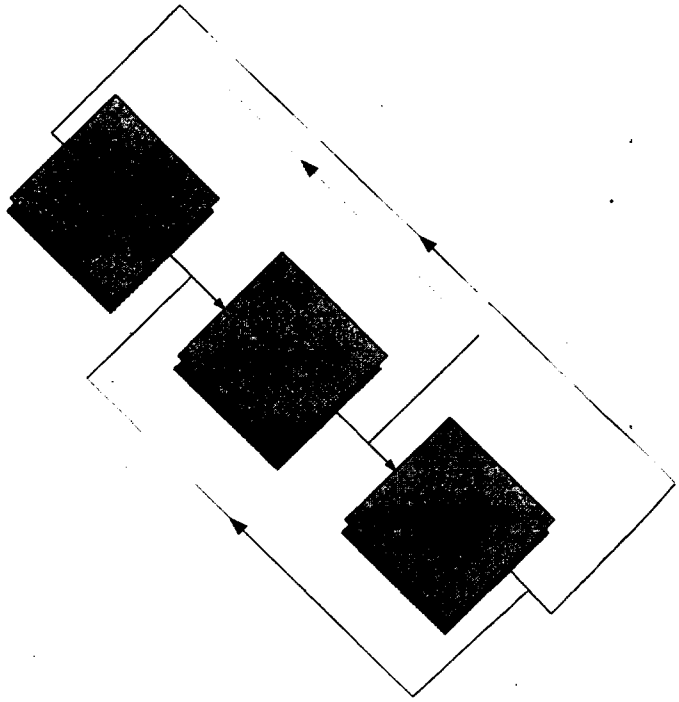


Figure 1.2 La synthèse des circuits traditionnelle

La deuxième étape est celle de la synthèse logique où la description obtenue à l'étape précédente est transformée en une représentation optimisée au niveau des portes logiques. Ces informations sont représentées sous la forme d'une liste d'interconnexions («*netlist*») qui décrit les portes logiques et ce qui les relie.

Enfin, lors de la synthèse physique, les opérations de placement et de routage permettent de produire les plans du circuit intégré qui seront utilisés lors de sa fabrication. Le placement permet de positionner les composants du circuit afin de répondre aux spécifications. Quant au routage, il permet de relier les composants entre eux. Cependant, il arrive fréquemment qu'une fois la synthèse physique terminée, les spécifications de départ ne soient pas rencontrées. Un retour aux étapes précédentes est alors requis, souvent plus d'une fois, pour pouvoir corriger la situation, ce qui coûte un temps précieux [Bryant *et al.*, 2001]. La figure 1.2 illustre bien les 3 étapes de la synthèse des circuits et le principe d'optimisation.

L'optimisation est une opération qui s'exécute lors de la synthèse. Elle permet principalement d'améliorer la performance des circuits. Afin d'obtenir une meilleure performance, des changements seront apportés aux circuits de telle sorte qu'il y aura une minimisation des délais. Les délais sont l'élément principal lorsque vient le temps d'optimiser, mais d'autres facteurs peuvent également agir sur la performance des circuits comme la superficie, la

puissance consommée, la testabilité et la robustesse face au bruit, pour n'en mentionner que quelques-uns. La superficie est représentée par le nombre de portes logiques et la taille de celles-ci. En règle générale, les délais sont inversement proportionnels à la superficie.

1.2.2 Les inconvénients de la miniaturisation

La miniaturisation a toutefois un prix. Depuis plusieurs années, des problèmes ont fait surface et ont dû être surmontés afin de pouvoir respecter la loi de Moore. En 1974, Robert Dennard avait déjà établi des règles qui permettent de conserver la performance des transistors lors de la miniaturisation [Dennard *et al.*, 1999]. Selon ces règles, la puissance exigée par transistor demeure inchangée. Au niveau du circuit, la puissance et la dissipation de chaleur requises augmentent donc au même rythme que le niveau d'intégration.

Par contre, l'effet des fuites de sous-seuil (en anglais, «*sub-threshold leakage*») des transistors sur la puissance totale du circuit n'a pas été pris en compte dans les calculs de M. Dennard [Bohr, 2007]. Dans les années 70, ces fuites étaient relativement faibles et l'impact sur la puissance totale consommée par le circuit était minime, mais ce n'est plus le cas aujourd'hui. En diminuant la taille des transistors, on a tendance à diminuer leur tension de seuil, ce qui a pour effet d'augmenter de façon exponentielle les fuites de sous-seuil [Blaauw *et al.*, 2004].

Des problèmes reliés aux interconnexions font également surface lors de la miniaturisation des circuits. À mesure que le niveau d'intégration augmente, l'importance relative de la capacité et la résistance des interconnexions augmente par rapport à celle des transistors, ce qui déplace la source principale de délais des portes logiques vers les interconnexions [Ho *et al.*, 2001].

La miniaturisation des composants entraîne indirectement des effets importants au niveau des interconnexions entre les composants. Malgré une diminution des délais en général, il y a une hausse des délais dans les interconnexions relativement aux délais dans les transistors. Ces délais dans les interconnexions ont un impact majeur lors de la synthèse. En effet, ils ne peuvent être connus qu'au moment de la synthèse physique. Traditionnellement, les systèmes de synthèses de haut niveau et logique ne peuvent qu'utiliser des approximations de ces délais. En conséquence, la structure logique des circuits générés repose souvent sur des données incomplètes ou même erronées. Ce n'est qu'au moment de la synthèse physique que les délais réels deviennent connus. Ce n'est donc qu'à cette étape qu'il est possible de vérifier précisément si les contraintes de délai fixées au commencement de la synthèse sont respectées. Lorsqu'elles ne le sont pas, il est fréquent qu'une structure logique inadéquate

en soit la cause. Il devient alors impératif de refaire la synthèse logique et d'itérer ainsi jusqu'à ce que nous respections les contraintes de délais exigées. Ce processus itératif devient de plus en plus long et complexe à mesure que l'importance relative des délais migre des portes logiques vers les interconnexions.

1.3 Définition du projet de recherche et objectifs

Les systèmes de conception logique actuels gèrent difficilement les problèmes liés aux interconnexions. En fait, ils sont axés sur la réduction des délais dans les portes logiques plutôt que dans les interconnexions. C'est pourquoi une nouvelle approche sera explorée : celle d'une synthèse des circuits altérée. Celle-ci est différente de la synthèse traditionnelle pour ce qui a trait à l'ordre dans lequel les étapes sont exécutées : la synthèse physique, en particulier le placement des ressources d'interconnexions, sera effectuée avant la synthèse logique afin d'éviter les multiples retours en arrière qu'occasionne souvent les systèmes de conception actuels.

De plus, les systèmes de synthèse employés par les grands de l'industrie sont basés sur des langages (C et C++) qui sont moins flexibles et moins portables que les langages orientés objets modernes. De plus, ces systèmes rendent difficile l'intégration de modules subséquents.

Puisqu'il n'existe pas de plate-forme de synthèse orientée-objet qui offre la possibilité de modifier l'ordre des opérations, en particulier pour effectuer une partie de la synthèse physique avant la synthèse logique, le projet présenté ici propose la conception d'une nouvelle plate-forme de synthèse de circuits qui n'aurait pas ces limites. La mise au point d'un système complet de synthèse exigeant un travail très important, le travail entrepris dans le cadre de cette maîtrise a été concentré sur la mise au point d'une infrastructure de synthèse (logique et physique), qui supporte les opérations de base essentielles au développement d'un système global. En particulier, dans l'optique choisie où la synthèse physique précède la synthèse logique, il est impératif que le système supporte l'entrée et la sortie de circuits logiques, la restructuration hiérarchique et la gestion de budgets temporels. Pour atteindre ces objectifs, un système a été développé, qui effectue les opérations suivantes : lecture et écriture de circuits logiques hiérarchiques, représentation interne de ces circuits selon une approche orientée-objet, modification de la hiérarchie, représentation des modèles de délais des portes logiques et infrastructure de calcul rapide des délais dans les circuits.

Plusieurs plate-formes ont été créées au cours des dernières années, mais aucune d'entre elles ne permet de mettre l'accent sur les délais dans les interconnexions au tout début

du processus de synthèse automatisée. Alors, la problématique qui doit être résolue est la suivante : la conception d'une plate-forme de synthèse de circuits axée sur les délais dans les interconnexions, devant être orientée-objet, portable et permettant l'intégration de modules futurs, comme celui portant sur la gestion des budgets de délai, est-elle possible ?

Donc, pour tenter de répondre à la problématique, le principal objectif du projet est de concevoir une plate-forme de synthèse de circuits, à l'aide d'un langage orienté-objet permettant la portabilité et l'intégration modulaire, axée sur les délais dans les interconnexions. Cet objectif peut se diviser en 4 objectifs spécifiques.

1. Le système doit être capable de lire une description logique provenant de la synthèse comportementale. Puisque l'étape de gestion de budgets de délai se situe entre la synthèse de haut niveau et la synthèse physique dans le nouveau paradigme, il va de soi qu'on se doit d'utiliser les informations provenant de la synthèse de haut niveau. Cet objectif implique la création d'un langage de description de circuits logiques hiérarchique.

2. Ce système doit être capable de représenter le circuit afin de pouvoir travailler avec les informations obtenues par la description logique. Cet objectif implique la création d'un analyseur syntaxique produisant une description orientée-objet du circuit lu. Cela est fait avec l'aide d'un graphe, plus précisément avec un réseau booléen. La raison pour laquelle ce type de graphe a été choisi comparativement à un graphe orienté acyclique est que le réseau booléen facilite l'utilisation de la hiérarchisation dans le graphe, étant donné que le circuit donné peut contenir plusieurs niveaux de logique.

3. En parallèle avec la description purement fonctionnelle de circuits, le système doit supporter la représentation des contraintes temporelles (les délais) du circuit traité. Pour ce faire, différents modèles de délais doivent être supportés et un module de gestion de budgets de délai permettant le calcul et l'identification de l'ensemble des contraintes temporelles du circuit traité doit être attaché à la plate-forme principale.

4. Finalement, les résultats obtenus lors de la gestion de budgets de délai doivent démontrer la validité de la plate-forme de synthèse.

Les sections suivantes de ce mémoire couvrent les éléments suivants : tout d'abord, la grammaire développée pour représenter la logique sera présentée, de concert avec la structure des objets créés lors de la lecture de circuits décrits par la grammaire. Ensuite, les opérations de modification de la hiérarchie sont présentées. Ces opérations reposent sur la représentation interne des circuits et permettent l'adaptation de leur structure lors des

opérations de synthèse. Par la suite, toute l'infrastructure de support et de traitement des délais est présentée. Cet élément du système global est essentiel à la mise en oeuvre éventuelle d'un système de synthèse, où les délais forment la contrainte principale influençant les décisions d'optimisation. Enfin, la validation de l'ensemble de la plate-forme est effectuée à l'aide du modèle logique d'un micro-processeur 32 bits. À l'aide de cette description complexe, toute la chaîne de traitement de la plate-forme est validée, de la lecture de fichiers de description logique et de contraintes temporelles jusqu'à l'obtention des chemins critiques du micro-processeur.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

Les algorithmes de synthèse logique, celle-ci ayant lieu traditionnellement entre la synthèse de haut niveau et la synthèse physique, sont à la fois ceux qui sont les plus puissants pour modifier la structure des circuits numériques et ceux qui agissent principalement au niveau du délai des portes logiques. Traditionnellement, l'optimisation précise des délais d'interconnexion se fait principalement lors de l'étape de synthèse physique. Cependant, à ce moment, le circuit est beaucoup moins facile à modifier. Alors, puisque les systèmes de conception actuels gèrent difficilement les délais des interconnexions, une nouvelle approche sera explorée : celle d'une synthèse des circuits altérée. Celle-ci est différente de la synthèse traditionnelle pour ce qui a trait à l'ordre dans lequel les étapes sont exécutées. En d'autres termes, la synthèse physique sera traitée avant la synthèse logique afin d'éviter les multiples retours en arrière qu'occasionne souvent les systèmes de conception actuels.

L'objectif de ce travail de recherche est de réaliser une plate-forme de développement de synthèse, pour permettre la mise au point d'une nouvelle façon d'ordonner les étapes de synthèse. Cependant, ce nouveau système doit supporter toutes les opérations nécessaires pour la synthèse. L'un des aspects principaux des systèmes de synthèse est l'optimisation des délais. Ce chapitre présente donc en détail les algorithmes de calcul de délai et de gestion de budgets de délai, afin de bien identifier ce que la plate-forme doit offrir.

2.1 La gestion de budgets de délai

Comme cela a été expliqué auparavant, jusqu'à récemment, les efforts étaient concentrés principalement sur la réduction des délais à l'intérieur des composants pour afin d'obtenir une meilleure performance. Les délais dans les interconnexions, quoique minimes comparés aux délais dans les transistors dans les débuts des circuits intégrés, ont toujours été considérés. Cependant, l'optimisation des délais dans les interconnexions n'est vraiment efficace que lors du placement et du routage, là où la longueur précise des interconnexions est connue, alors que les algorithmes d'optimisation des délais dans les transistors font leur travail principalement pendant l'étape de la synthèse logique. Ces algorithmes, bien qu'efficaces pour optimiser les délais dans les portes logiques, sont peu précis lorsqu'on

considère les interconnexions à ce niveau puisque les caractéristiques de ces dernières sont floues pendant la synthèse logique.

Alors, les multiples miniaturisations de ces composants au cours des années ont fait en sorte que les délais dans les interconnexions sont maintenant devenus la source prédominante de délais dans les circuits intégrés comme on peut le voir sur la figure 2.1. Selon [Keutzer *et al.*, 1997], en 1997, les délais dans les interconnexions représentaient environ 65% du délai global comparativement à seulement 10% pour les délais dans les composants. Depuis, la situation n'a cessé d'évoluer vers une différence encore plus marquée entre ces deux types de délais. Nous pourrions dire que les délais principaux ne sont plus dans le traitement des données, mais dans la transmission de celles-ci [Bryant *et al.*, 2001] [Cong, 1997] [Keutzer *et al.*, 1997]. Pour y arriver, il est important de prioriser la gestion des délais afin de s'assurer que le circuit respectera les contraintes temporelles.

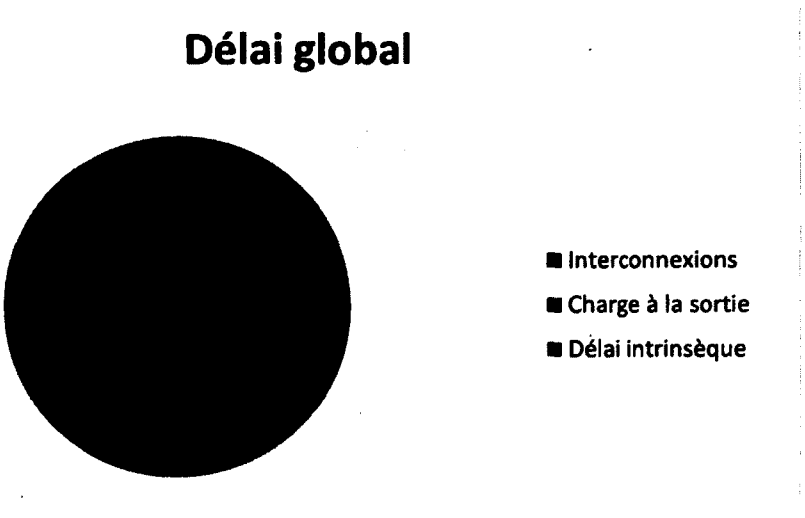


Figure 2.1 Proportion des délais dans un circuit

Le concept de gestion de budgets de délai lors de la synthèse des circuits est apparu pour la première fois [Ghiasi *et al.*, 2004] en 1983 lorsqu'on a voulu trouver une méthode plus efficace pour faire la compaction de circuits VLSI. En plus des contraintes dictées par le design, l'algorithme en question utilise également les contraintes fixées par l'usager, ce qu'on peut appeler des contraintes mixtes. Avec l'aide d'un graphe de contraintes où les budgets sont assignés aux nœuds, on tente de respecter les contraintes d'espacement dans un minimum d'itérations possible [Liao et Wong, 1983].

Il serait important d'expliquer ce qu'est la gestion de budgets de délai et de la situer par rapport aux trois principales étapes de la synthèse. Une explication simplifiée serait de dire que c'est la gestion des délais pour que le circuit rencontre les spécifications demandées.

Pour être plus précis, c'est la répartition de délais pour les différentes sections du circuit, dans le but d'attribuer le plus grand temps d'exécution possible à chaque composant, tout en respectant les contraintes de temps du système [Ghiasi *et al.*, 2004]. Ainsi, une fois cette étape terminée, une optimisation des composants est possible pour améliorer la superficie occupée et la dissipation de puissance. L'étape de la gestion des budgets de délai s'effectue généralement avant la synthèse physique puisque nous avons besoin d'informations, notamment les temps d'arrivée et les temps requis, fournies par la synthèse logique.

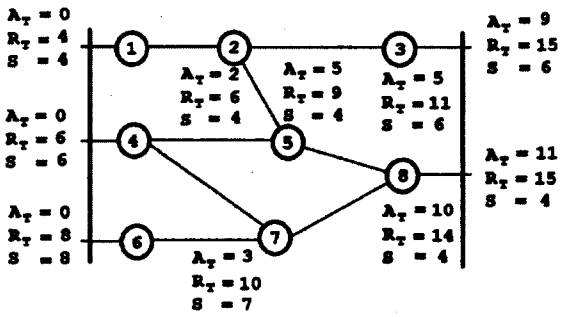
Il existe plusieurs façons de résoudre le problème de gestion de budgets de délai, la plupart d'entre elles utilisant des graphes, et les plus communes sont celles qui utilisent des graphes orientés acycliques (en anglais, «*Directed Acyclic Graph*», ou DAG) parce qu'ils offrent une bonne représentation d'un circuit. Un DAG est un graphe G composé de nœuds V et d'arcs E (nous avons conservé ici la nomenclature habituelle anglophone, qui dénomme les nœuds V pour «*vertex*» et les arcs E pour «*edge*»). Un délai est aussi associé à chaque nœud[1]existant. À ce graphe est associée une contrainte de temps globale à respecter. Sous cette contrainte, le budget est défini comme étant le délai maximum permis pour chaque nœud (ou arc) et chaque regroupement hiérarchique de nœuds, de façon à ce que la contrainte de temps globale soit respectée.

En ce sens, le budget de délai est très relié à la flexibilité (en anglais, «*slack*»). La flexibilité peut être définie comme étant la différence entre le temps requis et le temps d'arrivée. Les temps d'arrivée correspondent aux valeurs de temps déterminées par l'arrivée des signaux sur les points de contacts tandis que les temps requis correspondent aux valeurs de temps auxquelles les signaux se doivent d'être sur ces points de contact afin de respecter les contraintes temporelles. Lorsqu'on parle de budget de délais, celui-ci devient très important, car c'est cette valeur qui variera dans l'ajustement des délais entre les composants du circuit. Cette valeur de délai orientera et définira ce qui sera fait par la synthèse logique pour modifier le circuit en vue d'atteindre les objectifs de performance requis.

La valeur de la flexibilité indique la marge de manoeuvre disponible pour chaque noeud : une flexibilité positive indique qu'il est possible de ralentir le noeud en question sans impact sur le délai global, alors qu'une flexibilité négative indique que ce noeud se trouve sur un chemin dont le délai total dépasse la contrainte visée. Dans ce dernier cas, il est alors essentiel d'accélérer certains des noeuds se trouvant sur ce chemin. Cependant, dû

[1]Tout au long de ce document, les délais seront associés à un nœud, car cela est plus commun. Cependant, avec le système proposé, il peut aussi être associé à un arc.

à la dépendance entre les nœuds, la flexibilité totale des nœuds n'est pas identique au budget total que les nœuds peuvent tolérer [Bozorgzadeh *et al.*, 2003].



Nœud	Délai (ns)	A	B
1	2	4	-
2	3	-	-
3	4	2	6
4	3	6	-
5	5	-	4
6	2	7	1
7	4	1	7
8	1	-	-
Budget total	—	20	18

Figure 2.2 Exemple de gestion de délais

Un exemple de DAG est représenté à la figure 2.2. Sur ce DAG, (A_T), (R_T) et (S) sont évalués sur les nœuds. Ici, le temps d'arrivée (A_T), égal pour chaque entrée, est de 0 ns et le temps requis (R_T), égal pour chaque sortie, est de 15 ns, nous avons deux budgets possibles différents (A et B) appliqués sur le DAG, mais d'autres configurations de budget sont également possibles. Une fois le budget appliqué, il sera impossible aux autres nœuds de tolérer des délais supplémentaires tout en respectant la contrainte de 15 ns. Le budget A total est de 20 tandis que le budget B total est de 18. Ceci implique que le budget A est possiblement supérieur puisqu'il permet le plus de flexibilité (S) lorsque vient le temps de la conception. Il faut préciser, ici, que le budget total n'est pas égal à la flexibilité sur le circuit, mais bien au nombre d'incréments de délai alloués à l'ensemble des nœuds de façon à respecter les contraintes de temps.

2.2 Travaux existants sur la gestion des budgets de délai

L'incapacité de prédire de façon adéquate les charges que les interconnexions vont apporter au circuit et la façon dont ces charges seront disposées amènent un problème de conception : la finalisation temporelle du design. Ceci a amené les chercheurs à utiliser des modèles de charge pour prédire statistiquement les charges des interconnexions. L'utilisation de ces modèles fonctionnait assez bien pour des technologies plus grandes que 1 µm, mais des inconvénients sont apparus pour des technologies plus petites. Entre autres, la capacité des interconnexions et la résistance dans les couches de métaux jouent des rôles plus importants dans les technologies DSM (*«Deep Submicron»*) [Bryant *et al.*, 2001].

Alors, plusieurs recherches ont été effectuées sur la gestion de budgets de délai au cours des trois dernières décennies qui s'étendent sur plus d'un champ d'activité à des fins principalement d'optimisation. On peut voir son utilisation dans les diverses techniques reliées au respect des contraintes temporelles, lors de l'optimisation de la taille des composants dans le circuit et de l'optimisation de la puissance consommée par celui-ci, et même dans les étapes de placement et de routage lors de la synthèse physique.

Le respect des contraintes temporelles

Le problème du respect des contraintes temporelles (*«timing closure»*) est causé par l'incapacité de prédire adéquatement et avec précision les charges sur les interconnexions, et ce, avant la synthèse physique. [Bryant *et al.*, 2001] présente quelques techniques qui sont utilisées pour tenter de contourner ce problème.

1. Pour contrer l'incapacité de prédire les délais selon des modèles de charges (en anglais, *«wireload models»*), la synthèse avec le placement connu (*«placement-aware synthesis»*) est utilisée afin de faire une synthèse physique partielle qui sert à calculer les charges aux interconnexions en utilisant la prédiction de placement et l'estimation de routage. Cependant, la précision liée à cette prédiction dépend de la longueur des fils et de l'assignation de ces fils aux multiples couches du circuit intégré. Donc, il y a un manque d'informations reliées aux interconnexions avec seulement l'information sur le placement.

2. Avec la grosseur des blocs limitée pour la synthèse logique, on remarque que l'ère du DSM peut avoir recours à de nouvelles façons de faire la synthèse. En effet, une augmentation du nombre de composantes dans un circuit entraîne une augmentation de la grosseur des blocs ou bien un accroissement de la difficulté du problème de placement en raison du nombre d'interconnexions qui va exploser.

3. Dans une gestion des budgets de délai entre les blocs tels que les méthodes de [Tellez *et al.*, 1997] et [Nair *et al.*, 1989] utilisent, on alloue des budgets de temps à chaque sous-bloc du système incluant les interconnexions globales. Le problème est que la prédiction de délai pour les interconnexions globales est difficile, voire même impossible, avant que le circuit n'ait pris forme. La réallocation de la flexibilité est également coûteuse et très difficile pour chaque itération.

4. La synthèse avec délai constant qui s'appuie sur la notion que l'effort logique [Sutherland *et al.*, 1999] est le même pour chaque étage logique sur un chemin donné fonctionne avec des algorithmes ne travaillant pas sur la gestion des budgets de délai au niveau des blocs, mais à un niveau plus profond. L'effort logique est basé sur

le fait que différentes portes logiques comme l'inverseur, le NOR ou le XOR ont chacunes des caractéristiques propres pour actionner les charges capacitives. Cette procédure semble être une alternative viable seulement pour les problèmes utilisant les modèles de charge, car la capacité sur les interconnexions représente une bonne portion de la capacité totale des interconnexions.

5. Pour tenter de réduire le nombre d'itérations, la combinaison d'une planification au niveau des interconnexions et d'une synthèse avec délai constant a été proposée par [Otten et Brayton, 1998] et [Gosti *et al.*, 1998]. Le problème pouvant survenir dépend de la grosseur des blocs. Si les blocs ont une taille relativement petite de façon à ce que la synthèse avec délai constant fonctionne, la majorité des fils est globale et le placement physique des blocs sera critique face à la qualité de la planification des interconnexions, ce qui devient un problème de conception physique très ardu.

Le placement en fonction des délais et la planification par étage

Quelques travaux existent dans ce domaine de la synthèse. Tout d'abord, [Tellez *et al.*, 1997] ont tenté de faire le pont entre le placement et la gestion des budgets de délais avec un algorithme de placement basé sur les interconnexions allouant des modifications dans les deux domaines simultanément. Une réduction du chemin le plus long de 54 à 68% s'est vue comparativement aux autres algorithmes.

Ensuite, continuant les travaux de [Tellez *et al.*, 1997], un algorithme de gestion de budget de délai assurant un maximum de flexibilité lors du placement a vu le jour [Sarrafzadeh *et al.*, 1997]. Alors, en utilisant les travaux de [Gao *et al.*, 1991], le problème est formulé à l'aide d'une représentation sous la forme d'un graphe temporel. Ensuite, une conversion vers une gestion linéaire des budgets de délais se fait en approximant le problème de gestion convexe de budget de délai avec le problème de gestion linéaire par morceaux. De cette façon, une réduction de 50% en moyenne a été observée pour les violations des contraintes reliées à la longueur des interconnexions par rapport à l'algorithme de flexibilité nulle (ZSA pour «*zero slack algorithm*»). Une discussion plus élaborée sur cet algorithme est effectuée à la section 2.3.

Une unification de la gestion de budgets de délai et du placement a été proposée avec [Tellez *et al.*, 1997], mais des algorithmes extensibles pouvant faire cette unification restait encore un problème ouvert avant les travaux de [Kahng *et al.*, 2002]. Ceux-ci présument qu'il est possible de combiner l'optimisation des délais sur les interconnexions avec le placement «*top-down*», c'est-à-dire un placement commençant par les éléments les plus gros vers les

éléments les plus petits . De plus, il apporte une optimisation continue des délais sur les chemins qui est la première à éviter les heuristiques de gestion.

Enfin, puisque les approches existantes n'étaient efficaces qu'avec la logique combinatoire, [Yeh et Marek-Sadowska, 2003] ont proposé une solution pour pallier au problème de gestion de budgets de délais avec de la logique séquentielle. Il s'agit, en fait, de combiner les techniques existantes avec l'opération de «*retiming*» [Leiserson et Saxe, 1991] [Shenoy et Rudell, 1994]. Ainsi, une réduction des violations de contraintes de 16% et une amélioration sur le placement de 9% ont pu être observées.

L'algorithme de [Chen *et al.*, 2000], le MISA, peut être utilisé, entre autres, pour la planification par étages et pour le redimensionnement des portes. D'autres détails sont donnés sur cet algorihme à la section 2.3 puisqu'il s'agit d'un algorithme très utilisé.

L'optimisation des dimensions des composants et de la puissance

De nombreux travaux existent en ce qui a trait au redimensionnement des fils et des portes logiques et à l'optimisation de la puissance dans le circuit. Sous une contrainte de temps, la gestion permet de trouver dans un graphe d'interconnexions les nœuds dont les caractéristiques physiques, comme la taille ou la dissipation de puissance, peuvent être réduites en plaçant sur le circuit des pièces plus petites ou demandant moins de puissance, mais entraînant des délais plus grands [Ghiasi *et al.*, 2004]. Donc, pour tenter de réduire la puissance consommée, il est possible de changer les dimensions des portes logiques pour un circuit qui a déjà satisfait aux contraintes de temps [Mailhot et De Micheli, 1993] [Lin et Hwang, 1995], c'est-à-dire dont les contraintes de temps ne sont pas violées. Ceci est possible en identifiant les portes logiques que l'on veut modifier pour ensuite calculer la flexibilité en fonction du chemin.

Il est également possible de réduire la puissance consommée lorsque le circuit est contraint en temps, mais dont la satisfaction des contraintes n'est pas connue [Girard *et al.*, 1997]. Pour ce faire, un redimensionnement est fait sur les portes en fonction de la flexibilité sur chacune d'elle. La réduction de la puissance consommée variait entre 2,8 et 27,9% en comparaison avec les circuits ne faisant pas appel au redimensionnement.

Avec l'aide d'un paramètre de relaxation de délai, il est possible d'avoir une finalisation du circuit en redimensionnant les composants [Srivastava *et al.*, 2003]. La finalisation du circuit est atteinte lorsqu'aucune contrainte n'est violée et que le circuit est ainsi prêt pour la fabrication. Ce paramètre pourrait être aussi utilisé pour l'ordonnancement de la tension, par exemple. Le paramètre de relaxation de délai est également vu comme étant une nouvelle métrique de conception de circuit.

Il est également possible de faire de la gestion de budgets de délais, pour optimiser la puissance consommée, au niveau du transistor, c'est-à-dire à l'intérieur même des portes logiques [Kursun *et al.*, 2004]. Un graphe SP (série-parallèle) est utilisé pour trouver la solution et la complexité est de $O(n)$, où n est le nombre de transistors, ce qui en fait une solution optimale. Ce qui est bien, selon les auteurs, c'est que les principes trouvés sont des principes généraux.

Enfin, un modèle a été présenté pouvant traiter plusieurs politiques de budget de façon optimale [Ghiasi *et al.*, 2004]. Pour ce faire, une méthode combinatoire est utilisée pour résoudre le problème de budgets entiers, ce qui est différent de [Bozorgzadeh *et al.*, 2003] qui utilisait la programmation linéaire pour y arriver. Pour pousser les limites encore plus loin, une optimisation des fuites [Ghiasi, 2005] a été effectuée venant continuer les travaux de [Ghiasi *et al.*, 2004] en utilisant l'algorithme développé par [Bozorgzadeh *et al.*, 2003] dans ce contexte.

L'optimisation du placement et du routage

La compaction du dessin des masques (en anglais, «*layout*») cherche à diminuer la longueur des chemins existants dans le circuit, principalement celle du plus long chemin. Pour commencer, le concept de budget a été proposé en premier pour un problème de planification spatiale dans une compaction du layout [Liao et Wong, 1983]. Un graphe utilisant des contraintes mixtes, donc des contraintes minimales et maximales, est utilisé pour réduire la taille du circuit. Les travaux de [Lee et Tang, 1987] vont dans la même veine en ajoutant, en plus des contraintes mixtes, les contraintes de grille, qui sont les contraintes de positionnement des composants dans le circuit.

De plus, des percées ont été effectuées dans le domaines des circuits intégrés analogues [Felt *et al.*, 1992]. Des graphes sont utilisés pour résoudre le problème d'espacement pour ensuite utiliser, avec la solution comme point de départ, un algorithme de programmation linéaire pour résoudre le problème avec des contraintes de symétrie. Les contraintes de symétrie sont utilisées pour minimiser les décalages électriques et le bruit dans le circuit.

Enfin, l'algorithme proposé par [Ghiasi *et al.*, 2006] cherche à optimiser le placement et le routage pour des applications temps-réel, mais il sera discuté plus en détails à la section 2.3.

La planification au niveau des interconnexions

Afin de tenter de réduire les délais dans les interconnexions, les travaux de [Singhal *et al.*, 2007] ont été basés sur les travaux de [Ghiasi *et al.*, 2004]. Étant donné que le budget

de délai sur chaque nœud ne peut être augmenté à moins de violer les contraintes de temps (flexibilité nulle), le résultat obtenu est un design contenant beaucoup d'arêtes critiques. Donc, un algorithme de réduction des arêtes critiques (CER pour *«Critical Edge Reduction»*) va minimiser le nombre de fils critiques avec une solution de budget de délai maximal respectant des contraintes de délais fixes, ce problème étant NP-difficile.

Des travaux ont également été effectués par rapport à la dualité entre le délai dans les interconnexions et le délai dans les portes logiques [Hwang *et al.*, 2009]. Cette dualité est représentée comme suit : la capacité de l'entrée de la porte logique agit comme un filtre passe-bas diminuant la pente du signal ce qui affecte le délai dans l'interconnexion et la connexion ajoute une charge capacitive supplémentaire à l'entrée de la porte ce qui augmente le délai de propagation dans celle-ci. Donc, cette dualité est considérée afin d'obtenir une estimation précise des délais dans un circuit.

Enfin, la prédictabilité, ou l'action de pouvoir prédire efficacement, a été utilisée à quelques reprises dans les dernières années. La prédictabilité utilisée tôt dans le processus de conception, selon [Manohararajah *et al.*, 2006], a deux avantages. Le premier est que l'opération de restructuration liée aux contraintes de temps est plus efficace si le délai des interconnexions peut être prédit avec précision. Le deuxième est qu'une rétroaction peut être faite lors de la restructuration au lieu d'attendre après les longues étapes de placement et de routage pour y avoir droit. De plus, Le «persistance», employé par [Saxena, 2009], vise à réduire l'imprédictabilité lors du routage en attribuant à certaines interconnexions, sélectionnées avant le routage, des budgets de délais, en se basant sur le délais de l'interconnexion et les délais parasites, en vue des opérations d'optimisation subséquentes.

Alors, plusieurs éléments importants ressortent de ces travaux afin de mieux définir les propriétés que le système de synthèse doit posséder. Entre autres, l'utilisation des caractéristiques électriques des composants comme la résistance, la capacité et l'inductance apporte des informations nécessaires lors de l'optimisation des circuits. Les données sur la position des portes et des interconnexions sont utiles pour l'optimisation au niveau du placement et routage. La représentation hiérarchique des circuits permet de simplifier le processus d'optimisation. Il ne faut pas oublier l'utilisation de différents modèles de délai et les annotations multiples pouvant être faites sur les délais, tant sur les nœuds que sur les arcs. Le système de synthèse proposé intègre tous ces éléments.

2.3 Les algorithmes développés

Pour démontrer la flexibilité du système développé et aussi permettre la mise en oeuvre de budgets de délais, plusieurs possibilités se présentent. De nombreux algorithmes existent et le système doit être réalisé pour pouvoir les supporter et gérer les paramètres dont ils ont besoin. Quelques algorithmes importants sont maintenant présentés, pour identifier les paramètres nécessaires et assurer que le système développé puisse les supporter. Dans cette section, nous verrons quelques algorithmes qui ont été développés durant les 20 dernières années dans le domaine de la gestion de budgets de délais. La plupart des techniques utilisées dans ces recherches utilise des heuristiques non optimales comme l'algorithme sans flexibilité [Nair *et al.*, 1989] ou le MISA [Chen *et al.*, 2002]. Cependant, d'autres voies ont également été explorées. Pour la synthèse d'applications temps-réel, l'assignation des budgets de délais s'est vue exécutée selon un modèle probabiliste [Ghiasi *et al.*, 2006]. De plus, de récents développements portent à croire qu'un algorithme employant des délais entiers optimaux pourraient présenter des résultats intéressants [Bozorgzadeh *et al.*, 2003] [Ghiasi *et al.*, 2004].

2.3.1 L'algorithme sans flexibilité

Puisque les délais dus aux interconnexions devenaient de plus en plus présents dans les circuits VLSI, des méthodes et des algorithmes ont dû être introduits afin de palier à ce problème. L'algorithme sans flexibilité ou ZSA fut l'un des pionniers en termes de gestion de budgets de délai, principalement dans le domaine du placement et routage. Il garantit que la satisfaction des contraintes sur chaque interconnexion entraîne la satisfaction des contraintes de performance dans tout le circuit. Son existence est justifiée par les deux problèmes qu'il tente de résoudre. Le premier est que l'analyse temporelle est un processus itératif où le nombre d'itérations tend à augmenter avec la complexité des circuits. Le deuxième est que les informations relatives au temps changent à chaque itération, il se peut que la convergence se fasse lentement, voire même qu'elle soit inexistante [Nair *et al.*, 1989].

La méthode utilisée consiste à assigner des limites de longueur pour chaque interconnexion dans le circuit. Ainsi, la longueur des fils sur les chemins non critiques pourra être augmentée tout en s'assurant que la longueur des chemins critiques demeure sensiblement les mêmes. Cependant, un des éléments pouvant déterminer si un chemin est critique est le mode actif du signal, plus précisément si le circuit fonctionnera sur un front montant ou un front descendant. Pour pouvoir prévenir toute éventualité, le ZSA, avec quelques

modifications, peut être adapté pour l'un ou pour l'autre. Lorsque le circuit est actif sur un front descendant, le temps d'arrivée requis du signal est le temps le plus tardif que le signal peut se permettre d'arriver de façon à ce que le circuit respecte les contraintes. Inversement, nous pourrions dire que, pour un circuit actif sur un front montant, le temps requis est le plus tôt auquel le signal se doit d'arriver.

Avant de discuter de l'algorithme, il est nécessaire de définir ce qu'est la sécurité. Un circuit est dit sécuritaire s'il respecte les contraintes de temps dans les modes actifs sur fronts montant et descendant. Le ZSA permet de déterminer les délais qu'on peut se permettre d'ajouter aux composants dans le circuit de sorte que l'ajout de délais supplémentaires entraînerait au moins une sortie avec une flexibilité négative, donc non sécuritaire. Étant donné leurs différences, il faut traiter les deux modes actifs différemment.

Le mode actif sur front descendant est un peu plus simple à réaliser. Donc, dans une première étape, il calcule la flexibilité pour chaque composant du circuit. Ensuite, la plus petite flexibilité positive est sélectionnée pour être distribuée en tant que délai aux autres composants, devenant ainsi nulle. Ces étapes sont évidemment répétées jusqu'à ce que toutes les flexibilités soient nulles. L'algorithme est exécuté en un temps $O(np)$, n étant le nombre de composants dans le circuit et p, la grandeur de la liste d'interconnexions.

Lorsque vient le temps de traiter les signaux actifs sur front montant, l'algorithme précédent se doit d'être modifié quelque peu. En effet, il faudrait utiliser des délais incrémentaux négatifs pour obtenir les temps d'arrivée voulus, ce qui n'est pas très pratique. C'est pourquoi cet aspect est plus difficile à obtenir et le meilleur résultat n'est pas toujours garanti. L'algorithme utilisé dans ce cas, le quasi-ZSA, a un temps d'exécution semblable à celui décrit précédemment, soit $O(np)$.

Lorsque vient le temps de résoudre le cas où il faut tenir compte des fronts montant et descendant en même temps, la programmation linéaire est idéale, car il a été démontré qu'une solution en utilisant cette méthode peut être trouvée en un temps polynomial [Gács et Lovász, 1981]. Un ensemble minimal et un ensemble maximal de valeurs de délais sécuritaires sont définis, pour les modes actifs sur front montant et descendant respectivement, afin de créer un ensemble compatible puisque plusieurs valeurs de délais sont possibles sur les composants pour que le circuit soit sécuritaire. Une fois le système d'inégalités résolu, il faut trouver les ensembles respectifs avec le ZSA ou le quasi-ZSA.

Pour obtenir une réponse optimale au problème du ZSA, donc pour le cas où le mode est actif sur front descendant, une résolution du problème sous une forme linéaire sera utilisée, sans les éléments touchant le mode actif sur front montant. En se servant d'un ensemble

minimum de droites, un système d'équation permet de trouver l'enveloppe ayant une inclinaison maximale, celle-ci étant la pente la plus élevée. Une fois cette étape accomplie, le ZSA peut être invoqué afin de trouver un résultat optimal.

Étant donné que le ZSA était une innovation, il n'était pas sans faiblesses. C'est pourquoi quelques algorithmes ont été proposés se basant sur celui-ci tout en l'améliorant, mais deux d'entres eux sont particulièrement importants. Le premier qui vient corriger quelques lacunes du ZSA est l'algorithme Minimax [Youssef et Shragowitz, 1990]. Malgré le fait qu'il cherche toujours à obtenir une flexibilité nulle dans le circuit, cet algorithme se distingue de son prédécesseur en trois points. Le premier est qu'il possède une complexité temporelle linéaire au lieu d'une de niveau quadratique pour le nombre d'interconnexions. Le deuxième est que la distribution des flexibilités est faite de façon proportionnelle à la distribution des délais de chargement des entrées plutôt que d'avoir une distribution uniforme. Enfin, le modèle de délai utilisé est plus détaillé que celui utilisé par le ZSA.

Le deuxième algorithme pour améliorer le ZSA a été proposé par [Gao *et al.*, 1991] et il est axé sur la performance lors du placement. À l'aide d'un algorithme de programmation convexe, il détermine la longueur maximale des interconnexions. Ensuite, il choisit un ensemble de limites supérieures ou «*upper-bounds*», celles-ci étant les contraintes de temps maximales qu'il faut respecter, ce qui permet d'obtenir un algorithme de placement ayant le maximum de flexibilité. Elles donnent aussi la possibilité de modifier l'ensemble de contraintes lorsque l'algorithme échoue un placement satisfaisant ces contraintes.

Finalement, [Kuo et Wu, 2000] cherchent à faire de la synthèse en respectant les contraintes de temps et montrent que le ZSA est encore utilisé, plusieurs années après son apparition. Pour ce faire, ils ont recours à deux algorithmes qui font appel au ZSA : l'algorithme de distribution de flexibilité équilibrée et l'algorithme de distribution de flexibilité basée sur le rapport aire/surface.

2.3.2 L'algorithme basé sur un nombre d'ensembles indépendants

Étant donné que tous ces algorithmes, le ZSA et ceux qui en sont dérivés, employés principalement lors de la conception de circuits VLSI, ne fournissent pas de solutions optimales, un nouvel algorithme basé sur un nombre maximum d'ensembles indépendants ou MISA est apparu pour remédier à la situation [Chen *et al.*, 2000]. Le MISA est une version améliorée du ZSA et permet de mieux optimiser le circuit en tenant compte de la flexibilité. Pour cela, il utilise les chemins non-critiques, donc ceux ayant un potentiel pour la réduction du ratio surface/puissance. De plus, comme vu précédemment, cet algorithme a trouvé

application dans l'optimisation du dimensionnement des portes et dans le placement en fonction des délais («*timing-driven placement*»).

Les auteurs apportent la notion qu'il existe deux formes de budget : statique ou dynamique. Dans un budget statique, la sensibilité de la flexibilité vient jouer un rôle important. Par cela, on entend que la flexibilité attribuée à un nœud va diminuer s'il y a ajout d'un certain délai à ce nœud. Quant au budget dynamique, c'est le gradient budgétaire qui prime. Celui-ci est un bon indicateur lorsque vient le temps de déterminer si un plus grand budget est possible si la contrainte de temps est relâchée [Chen *et al.*, 2002]. Ensuite, il y a la sensibilité de la flexibilité qu'il faut considérer. En considérant deux nœuds, ou arcs, adjacents, on dit qu'un nœud est sensible à la flexibilité si un incrément de délai sur l'autre nœud entraîne une diminution de la flexibilité sur ce même nœud.

La résolution d'un problème axé sur nombre maximum d'ensembles indépendants est un problème NP-complet pour la majorité des graphes. C'est pour cela qu'une transformation est nécessaire pour obtenir un graphe transitif avec égalisation des flexibilités, lui donnant ainsi un temps d'exécution en un temps polynomial $O(K * n^3)$, où K est le nombre de flexibilités distinctes et n est le nombre de nœuds dans le graphe.

L'algorithme MISA peut être porté vers un DAG avec poids en le modifiant en un algorithme basé sur le nombre maximal d'ensembles pesés indépendants ou MWIS. Le poids d'un nœud peut être une mesure de l'importance du budget de délai de ce nœud vis-à-vis les autres. Le temps d'exécution de l'algorithme est d'ordre $O(n*\log n)$ où n est le nombre de nœuds. Enfin, contrairement au ZSA, le MISA est plus performant en présence d'un grand nombre d'éléments à la sortie, ou «*fanouts*». Par contre, malgré qu'il soit 30% plus efficace que le ZSA, il ne parvient pas, lui non plus, à être optimal en tout temps. L'heuristique présentée n'est optimale que lorsque les effets sont minimes ou nuls entre chaque étape lors de son exécution.

2.3.3 L'algorithme se basant sur les délais entiers dans un DAG

Le ZSA et le MISA sont tous les deux très efficaces pour la gestion de délais et c'est pour cela qu'ils sont fréquemment utilisés. Par contre, ces deux algorithmes ne donnent pas toujours une solution optimale. C'est pour cette raison qu'un groupe de chercheurs se sont penchés sur la question en tentant de solutionner le problème de la gestion de budgets de délai entiers, un problème qui existe depuis 1983 [Liao et Wong, 1983]. Lorsqu'il est question de délai entier, il s'agit évidemment d'un délai non fractionné. Donc, cet algo-

rithme utilisera la programmation linéaire et des délais entiers pour obtenir une solution optimale.

Il existe deux raisons principales pourquoi il serait préférable de travailler avec des délais entiers. La première est que les librairies utilisées comprennent des composants dont les caractéristiques de délais sont exprimées sous la forme d'entiers. Par exemple, le délai pour certains composants peut être exprimé en termes de cycles d'horloge. La deuxième raison est que les solveurs de programmation linéaire sont souvent instables et ils ont de la difficulté à converger vers une solution en raison de la propagation des erreurs d'arrondi sur la partie fractionnaire des nombres réels. Les algorithmes ZSA et MISA peuvent être modifiés pour prendre en compte des délais entiers, mais il ne sera pas possible de prédire si les résultats obtenus seront optimaux [Bozorgzadeh *et al.*, 2003].

Lorsqu'il est question de programmation linéaire, il existe quelques cas où la solution trouvée est implicitement optimale. C'est le cas pour les problèmes de flux de réseaux. Avec l'aide d'une relaxation sur la programmation linéaire, les problèmes de programmation linéaire entière (ILP) peuvent être résolus de façon optimale, car le graphe en entrée est un chemin orienté. Par contre, un DAG ne remplira pas toujours cette condition. Pour s'assurer de l'optimalité de la solution, une nouvelle assignation du budget sera effectuée pour ensuite être appliquée sur la solution donnée par la programmation linéaire. Le temps d'exécution de cet algorithme est d'un ordre polynomial $O(|V^2|)$, où V est le nombre de nœuds dans le DAG. De plus, il faut remarquer qu'il surpasse de façon significative les performances du ZSA.

2.3.4 Le système axé sur le modèle probabiliste

L'assignation d'un budget de délais basé sur un modèle probabiliste est une avenue très récente qui a été employée pour la synthèse d'applications en temps-réel léger, ce qui inclut l'encodage vidéo, la compression d'image et le playback audio. Le terme «léger» signifie que le non-respect des contraintes temporelles est admis pour des entrées de données non fréquentes [Ghiasi *et al.*, 2006].

Pour employer ce modèle, un graphe de flux de données de contrôle (CDFG) est utilisé. Ce type de graphe est composé de nœuds qui sont eux-mêmes des graphes de flux de données à l'interne. Donc, à chaque nœud est attribué un délai et tous les chemins possibles dans le nœud doivent rencontrer obligatoirement la contrainte de délai du nœud en question. Enfin, les probabilités qu'un nœud soit emprunté plutôt qu'un autre sont connues a priori.

L'algorithme cherche à augmenter le délai ou plutôt d'assouplir la contrainte de délai. Pour y arriver, les nœuds ayant une probabilité de visite élevée se verront attribués un délai plus petit. Quant aux nœuds peu fréquentés, ils peuvent se voir assigner un délai plus élevé sans qu'il y ait un grand impact sur le délai global de l'application. Le problème de budget de délais dans les CDFG est séparé en deux étapes : l'assignation à chaque nœud du budget de délai supplémentaire et, par la suite, la distribution de ce budget à l'intérieur du nœud. En terme de complexité temporelle, le problème de gestion des délais dans un CDFG est équivalent au problème du sac à dos, qui a été prouvé NP-complet. Pour plus d'informations au sujet du problème du sac à dos et d'autres problèmes NP-complet, se référer à [Cormen *et al.*, 2001]. Grâce à cet algorithme, des gains de 26% sur la surface utilisée ont pu être observés et ceux-ci sont supérieurs au principal compétiteur, un algorithme de gestion de budgets de délai optimal utilisant des contraintes en temps-réel.

En résumé, la plate-forme de synthèse doit incorporer plusieurs éléments provenant de ces travaux. En effet, elle supporte les temps de montée et de descente qui sont exigés par le ZSA et la représentation hiérarchique, exigée par les CDFG. La représentation interne du circuit par un réseau booléen, un type de graphe, est également exigée par quelques algorithmes comme le ZSA et le MISA. La possibilité d'utiliser plusieurs modèles de délais (linéaire, linéaire par morceaux, etc.) est requise pour chacun des algorithmes. Le chapitre suivant présente le langage de description logique hiérarchique ainsi que le modèle interne créé pour la représentation du circuit en mémoire. Ce modèle intègre les éléments essentiels identifiés dans ce chapitre, soit la hiérarchie, l'utilisation d'un réseau booléen pour représenter le circuit, les modèles de délai, les annotations, les temps de montée et de descente et les caractéristiques électriques des composants.

CHAPITRE 3

MODÉLISATION DU CIRCUIT LOGIQUE

Pour résoudre le problème de la convergence temporelle («*timing closure*»), le système proposé doit permettre de modifier l'ordre habituel des étapes de synthèse. Comme mentionné précédemment, la synthèse de circuits selon un nouveau paradigme est envisagée, un paradigme qui exige que la synthèse physique soit exécutée avant la synthèse logique, puisque les délais ne sont plus majoritairement concentrés dans les composants, mais se trouvent plutôt dans les interconnexions. L'étape de planification temporelle, elle, se situe entre la synthèse de haut niveau et la synthèse physique. Cette étape est très importante, car elle permet d'associer un budget aux différents sous-circuits pour améliorer la performance.

Tout d'abord, la synthèse de haut niveau doit fournir quelques informations nécessaires pour que l'expérience soit réalisable. Donc, les informations reçues comprendront nécessairement les temps d'arrivée, les temps requis et les blocs internes du circuit. Dans l'ordre proposé, la synthèse de haut niveau précède la synthèse physique, ce qui oblige celle-ci à demander certaines informations qui, habituellement, sont fournies par la synthèse logique. Que l'on voit la planification temporelle comme étant la dernière étape avant la synthèse physique ou la première étape de celle-ci n'a peu d'importance puisqu'elle doit recevoir les mêmes informations pour être en mesure de bien s'exécuter.

Figure 3.1 La synthèse des circuits

Il est important que la structure logique et hiérarchique du système puisse être partagée. Toutes les informations relatives aux délais et à leur calcul sont essentielles. Ce chapitre se concentre sur les éléments de la nouvelle plate-forme qui supportent ces éléments, c'est-à-dire la grammaire de description hiérarchique et temporelle de circuits numériques, ainsi que la représentation interne de ces structures dans la plate-forme. La figure 3.1 illustre

le flux de traitement avec, au centre, les parties importantes de la plate-forme, soit la grammaire, la mise en place du réseau booléen et l'implémentation de fonctions utiles à la synthèse logique. On peut également remarquer que la gestion de budgets de délai fait partie de la synthèse physique et que le tout forme l'infrastructure développée.

L'information transmise par la synthèse de haut niveau est transmise sous la forme d'une description logique. Il faut alors extraire toute l'information nécessaire de cette description afin de pouvoir représenter le circuit et, éventuellement, faire de la gestion de budgets de délai. Pour y arriver, il est nécessaire de lire cette description logique et de traiter l'information s'y trouvant.

3.1 Grammaire

Il a été expliqué précédemment que le processus de synthèse proposé suivra un nouveau paradigme. Dans l'approche traditionnelle, l'étape de synthèse logique détermine en partie les délais et les contraintes temporelles. Or, les contraintes temporelles doivent provenir du système de synthèse de haut niveau avec la nouvelle approche puisque la synthèse physique précède la synthèse logique. La description logique, provenant de la synthèse de haut niveau, représente le comportement d'un circuit électronique à l'aide d'un ensemble de fonctions et d'équations. Ce travail de recherche étant situé au moment où les bases du système doivent être établies, la première étape est de permettre l'échange d'informations entre le système de synthèse de haut niveau et le système de synthèse physique. Dans ce but, un langage de description logique a été défini et celui-ci est présenté dans la section qui suit.

De façon à pouvoir modéliser adéquatement une description logique provenant de la synthèse de haut niveau, il était nécessaire d'avoir un outil nous permettant de le faire. Puisque la description logique se retrouve dans un fichier et que son contenu n'est pas chiffré, la réponse à ce problème était évidente : une décomposition analytique était la voie à suivre. Cependant, deux options étaient disponibles : faire la conception complète d'un analyseur syntaxique ou utiliser un analyseur existant en l'intégrant dans notre programme. Comme la théorie des compilateurs est établie depuis longtemps, il existe de nombreux outils qui sont conçus pour accélérer la création d'analyseurs syntaxiques et lexicaux. C'est donc cette deuxième option qui a été choisie, ce qui a permis de mettre l'accent sur la qualité et l'expressivité de la grammaire plutôt que sur le code permettant de la mettre en oeuvre.

Ainsi, le générateur de décomposeur analytique ANTLR[1] mis au point par Terence Parr [Parr, 2007], professeur à l'université de San Francisco en Californie, a été retenu pour sa facilité d'utilisation, pour sa courbe d'apprentissage rapide, pour son coût inexistant et pour sa capacité d'intégration à la plate-forme de développement Eclipse. ANTLR simplifie grandement la vie de l'usager parce qu'il génère lui-même les fichiers responsables de la décomposition analytique et de l'analyse lexicale. Pour cela, un ensemble de règles a été conçu afin de permettre à ANTLR de générer des fichiers capables de lire une description logique. Ces règles ont été rédigées en fonction des caractéristiques nécessaires de la description logique telles qu'identifiées au chapitre précédent. L'ensemble de ces règles, appelé «grammaire», est présenté dans ce qui suit.

Dans une grammaire, les règles déterminent l'action à entreprendre lorsque l'expression qui la définit est rencontrée. Pour cela, chaque mot lu est stocké sous la forme de jeton («token») et la règle choisie dépend de l'agencement de ces jetons. Pour traiter plus facilement l'information, ANTLR permet de mettre la grammaire sous la forme d'arbre de syntaxe abstrait ou AST («*Abstract Syntax Tree*»). Étant donné que l'analyseur syntaxique en arbre est sensiblement identique à l'analyseur syntaxique, il ne sera pas couvert lors de l'explication de la grammaire.

C'est dans la grammaire en arbre que l'on commence réellement à traiter l'information provenant de la description logique. Les actions pour chaque règle consistent à prendre l'information désirée et la stocker à l'endroit voulu. Or, pour conserver un niveau d'indirection entre la grammaire et la plate-forme de conception et, ainsi, séparer les deux entités, un pont unidirectionnel a été érigé. Ce pont prend toutes les requêtes provenant de la grammaire et exécute les actions dans la plate-forme. En d'autres termes, c'est elle qui crée le réseau booléen.

Le réseau booléen est un graphe acyclique orienté (en anglais, «*Directed Acyclic Graph*», ou DAG) où les nœuds sont principalement des portes logiques. Or, le réseau booléen construit est légèrement différent puisqu'il doit supporter la hiérarchie. Ceci amène la particularité que les nœuds peuvent aussi bien être des portes logiques que des instances de fonction. La figure 3.2 illustre le réseau booléen qui sera employé dans le système. La fonction contient trois entrées, deux sorties, deux équations et une instance de fonction. La hiérarchie est présente au niveau du nœud de l'instance, qui contient à lui seul tous les éléments de la fonction instanciée.

[1] «ANother Tool for Language Recognition» ou «un autre outil pour la reconnaissance du langage»

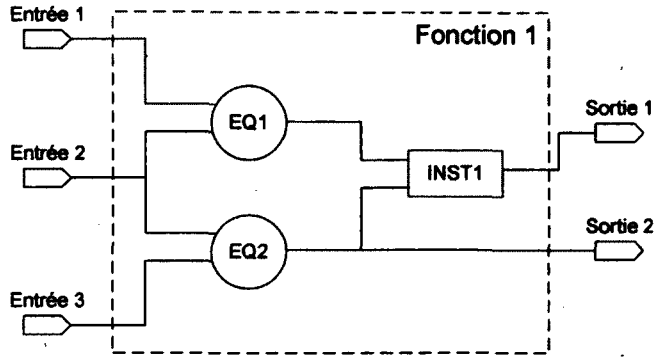


Figure 3.2 Le réseau booléen hiérarchique

Pour bien réussir à modéliser un circuit logique, deux grammaires différentes ont été nécessaires : l'une pour la description logique et l'autre pour l'ensemble des bibliothèques de portes logiques. Ces bibliothèques contiennent des informations supplémentaires comme les capacités aux entrées et aux sorties et les modèles de délais utilisés. Plusieurs règles sont les mêmes pour les deux grammaires puisque certains éléments ne sont pas uniques à une grammaire. Cependant, pour éviter un éventuel conflit entre les règles, il a été décidé de créer deux grammaires distinctes définies séparément.

3.1.1 Grammaire liée à la logique

La description logique, comme mentionné précédemment, représente le comportement d'un circuit électronique comportant plusieurs fonctions et équations. L'extension utilisée pour les fichiers contenant la description liée à la logique du circuit est *.llog*, extension qui est différente des fichiers utilisés pour la description liée aux modèles. Celle-ci est construite de façon à ce que quiconque puisse la lire et la comprendre intuitivement. Puisque la simplicité est préférable à la complexité, quelques particularités en font partie qui présentent ce principe, la première étant la présence d'instances. Celles-ci permettent de mieux supporter la hiérarchisation dans le système. Par exemple, il est plus simple et convivial de décrire un additionneur 32 bits à l'aide de 32 instances d'additionneur 1 bit, plutôt que d'utiliser une description unique qui regroupe l'ensemble de la fonctionnalité logique. En même temps, cela réduit largement la taille de la description logique. De plus, cela permet aux outils de synthèses physique et logique d'utiliser la hiérarchie pour mieux optimiser le circuit. Par exemple, lors de la synthèse logique, il est possible de n'optimiser qu'un bloc de niveau inférieur et ensuite d'en utiliser les instances.

Il est important que les principaux opérateurs logiques (**ET**, **OU**, **NON**) soient supportés au niveau des équations, car ce sont des équations de base. Cependant, avec quelques ajustements mineurs, d'autres opérateurs comme le **XOR** pourraient être, eux aussi, supportés. Pour l'instant, ces opérateurs n'existent que sous la forme d'une combinaison des principaux opérateurs. Une grande innovation qui est apportée avec cette description est la capacité de représenter toute porte séquentielle utilisant un opérateur distinct. Il est vrai qu'il existe plusieurs portes séquentielles différentes, chacune ayant sa raison d'être et pouvant être construite avec l'aide de portes logiques simples. Puisque le calcul des budgets de délais aux portes séquentielles est principalement relié à la période de l'horloge y étant attachée et pour raison de simplicité, toutes les portes séquentielles sont représentée par une équation ayant l'opérateur **@SEQ**. Les entrées de cette équation détermineront ainsi son comportement. Il en va de même pour les éléments de logique à trois états. Ces éléments fonctionnent parfois dans un état de haute impédance. Alors, il est nécessaire de pouvoir bien les représenter. Pour cela, l'opérateur **@TRI** sera employé.

L'analyseur syntaxique

Comme l'illustre la figure 3.3, les différents éléments composant la description logique sont classés par ordre d'importance, du haut vers le bas. Cette figure est une représentation hiérarchique de l'analyseur syntaxique se trouvant à l'annexe 3.4. En tout premier lieu, nous avons une règle maîtresse qui dit qu'un circuit est composé de plusieurs bibliothèques. Le terme «circuit» n'est ici que le nom d'une règle, donné de façon arbitraire, et n'implique pas un véritable circuit. Ensuite, chaque bibliothèque est définie par son nom suivi d'un ensemble de fonctions. Le nom de la bibliothèque peut être facultatif, ce qui implique qu'un nom arbitraire lui sera donné par la suite.

Chaque fonction est composée d'une en-tête suivie d'un bloc. Tout d'abord, l'en-tête est composée du nom de la fonction suivi des paramètres d'entrée, de sortie et d'entrée/sortie de celle-ci, dans cet ordre. Les entrées/sorties sont utiles lorsque les actions de lecture et d'écriture utilisent la même interconnexion, un accès à la mémoire par exemple, et elles sont toujours associées à des éléments de logique à trois états. Chacun de ces types de paramètres sont séparés par un point-virgule, même si la liste de paramètres est vide. Les paramètres d'entrée et d'entrée/sortie sont les seuls qui sont facultatifs. Quant au bloc, il est composé d'une série de déclarations.

On entend principalement par déclaration une instance ou une équation. L'instance, sur un niveau supérieur, est un lien avec une fonction située sur un niveau hiérarchique inférieur.

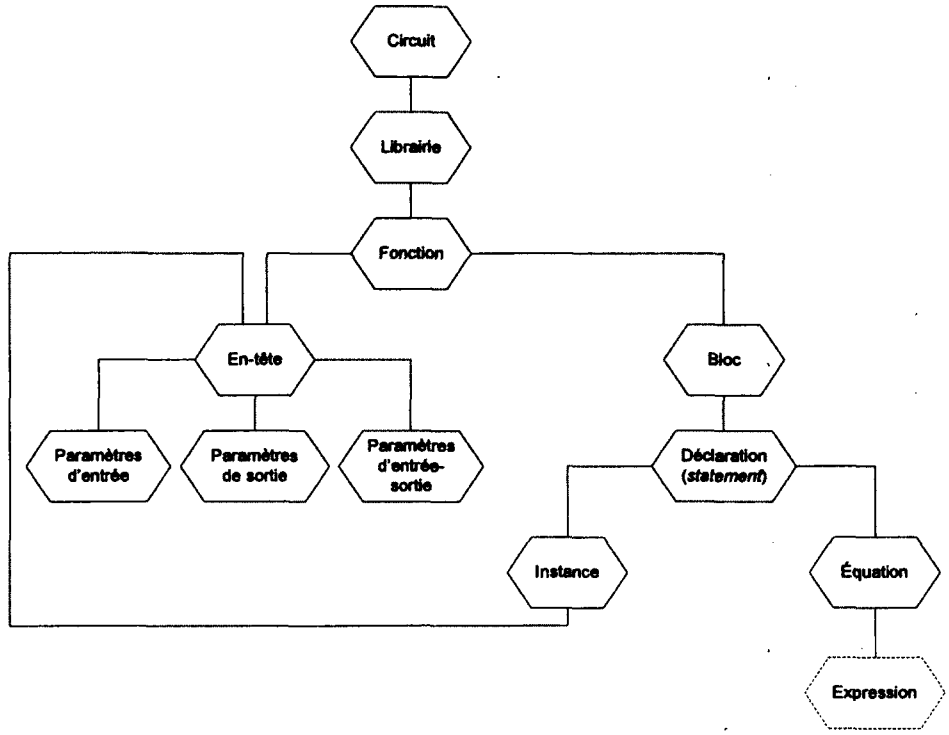


Figure 3.3 Hiérarchie des règles de l'analyseur syntaxique lié à la logique

En regardant la figure 3.4, nous remarquons qu'il existe quatre conditions pour que cette règle soit choisie. Dans l'ordre :

1. Il peut s'agir d'une instance, qui est donnée par son nom suivi par le nom de la bibliothèque, qui est optionnel, et la fonction à laquelle l'instance fait référence. Encore une fois, ici, lorsque le terme optionnel est manquant, un nom arbitraire est donné, préférablement le même que précédemment.

2. Il peut s'agir d'une équation, donnée par une sortie égalant une expression logique.

3. Il peut s'agir d'éléments servant à la gestion des budgets de délai comme l'initialisation des valeurs de temps d'arrivée aux entrées et de temps requis aux sorties d'une fonction, et ce, en montée et en descente.

4. Enfin, il peut s'agir de l'initialisation des périodes d'horloge pour les portes séquentielles se trouvant dans la fonction.

```
circuit: library+ ;

library: ('LIB' ID)? '{' function+ '}' ;

function: user_def bloc ;

bloc: '{' stmt+ '}' ;

user_def: ID '(' inparam? ';' outparam ';' ioparam? ')' ;

inparam: ID (',' ID)* ;

outparam: ID (',' ID)* ;

ioparam: ID (',' ID)* ;

stmt: ID (ID '.')? user_def SIMT_END |
      ID '=' expr SIMT_END |
      'set' TIME TRANSITION ID ':' VALUE SIMT_END |
      'PERIOD' ':' VALUE SIMT_END ;

expr: multExpr ('+' multExpr)* ;

multExpr: phase (multop phase)* ;

multop: MULT | ' ' ;

phase: atom (INV)? | '!' atom ;

atom: ID | pre_def | '(' expr ')' ;

pre_def: SEQ '(' param_seq ')' | TRI '(' param_tri ')' ;

param_seq: ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ;

param_tri: ID ',' ID ;
```

Figure 3.4 L'analyseur syntaxique pour la logique

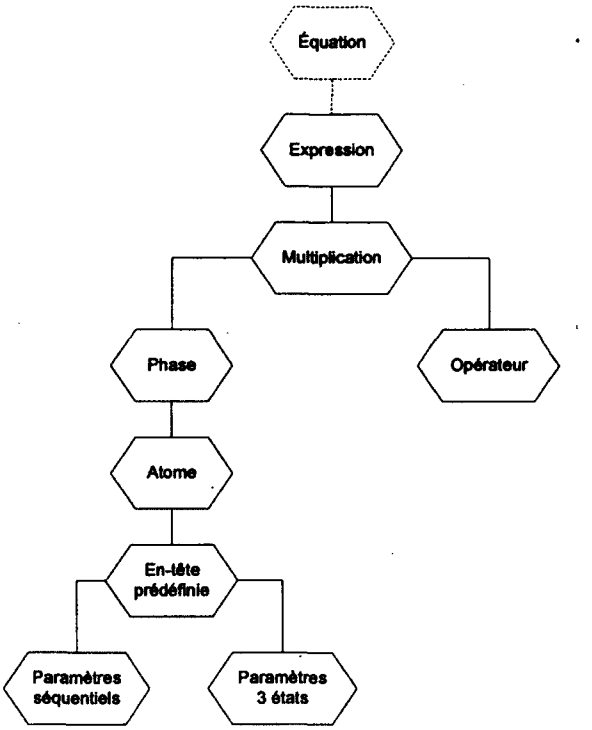


Figure 3.5 Hiérarchie des règles de l'analyseur syntaxique lié à la logique (suite)

En ce qui concerne les expressions logiques, elles doivent avoir la capacité de représenter tous les opérateurs logiques existants. La figure 3.5 présente la suite des éléments de l'arbre défini à la figure 3.3. Ici, le OU logique (+) a prédominance sur le ET logique (*) qui, lui, a prédominance sur le NON logique (!). Une règle spéciale a été construite afin de supporter l'espace entre deux variables comme s'il s'agissait de l'opérateur *. Il en va de même pour l'opérateur de négation qui est représenté par deux opérateurs : le ! qui est employé avant la variable et le ' qui est employé après celle-ci. Enfin, une expression logique peut être, à la base, une expression logique entre parenthèses, un élément de logique prédéfinie ou bien tout simplement une variable.

Finalement, les éléments spéciaux de logique prédéfinie sont représentés par un caractère spécial, soit **@SEQ** ou **@TRI**, suivi d'une liste de paramètres. Celle-ci est différente de celle provenant d'une fonction parce que le nombre d'éléments dans cette liste est connu dès le début. Pour un élément de logique séquentielle, cette liste comporte neuf éléments tandis qu'elle n'en comporte que deux seulement pour les éléments de logique à trois états.

Les paramètres pour la porte séquentielle sont, dans l'ordre, le signal de donnée (**D**), le «*Set*» synchrone (**SS**), le «*Set*» asynchrone (**AS**), la priorité du «*Set*» (**SP**), le «*Reset*» synchrone (**SR**), le «**Reset**» asynchrone (**AR**), la priorité du «*Reset*» (**RP**) et le signal

d'horloge (**CK**). Le neuvième élément ne fait que préciser s'il s'agit de la sortie Q ou de la sortie QB. Les portes séquentielles sont représentées ainsi afin de pouvoir représenter tous les éléments séquentiels possibles à l'aide d'une seule expression. Les deux entrées liées à la priorité déterminent laquelle des deux entrées, le «*Set*» ou le «*Reset*», est prioritaire. Un signal est synchrone lorsqu'il est régulé par le signal d'horloge tandis qu'un signal asynchrone peut être un signal non régulé par un signal d'horloge.

L'analyseur lexical

La figure 3.6 montre l'analyseur lexical pour la logique. Ici, les règles définissent la composition des jetons. Si on se réfère à la troisième condition de l'affirmation, on remarque les jetons «**TRANSITION**» et «**TIME**». Le premier se doit d'être «**RISE**» ou «**FALL**» sinon une erreur se produira lors de l'exécution. La plupart de ces règles sont évidentes et ne seront pas expliquées de façon spécifique. Cependant, les règles suivantes sont plus complexes et requièrent une courte explication :

- La règle «**TRANSITION**» indique si les signaux sont en montée ou en descente. L'importance de ce paramètre pour bien caractériser les délais et les contraintes temporelles est justifiée par le fait que les délais peuvent varier selon que les signaux soient en montée ou en descente.

- La règle «**TIME**» permet de spécifier si la valeur de temps devant être initialisée est un temps d'arrivée ou un temps requis

- La règle «**FLOAT**» définit un nombre à point flottant avec deux entiers séparés par un point.

- La règle «**BOOL**» représente un 0 ou un 1 logique qui entre directement dans le circuit.

- La règle «**STRING**» indique une chaîne de caractères, le premier n'étant pas un chiffre.

- La règle «**FLAT**» permet de représenter le nom d'une interconnexion après un aplanissement. Puisque plusieurs interconnexions peuvent avoir le même nom après cette opération, il a été convenu que le nom de la fonction instanciée serait ajouté à la suite du nom de l'interconnexion, ces deux noms étant séparés par un deux-points, un caractère qui n'est pas admissible par la règle précédente. Il est à noter que la hiérarchie est supportée et est affichée à l'intérieur de ce nom.

- La règle «**WS**» représente les espaces blancs qui ne doivent pas être pris en compte. Cela inclut également la tabulation, le retour de chariot et un changement de ligne.
- La règle «**COMMENT**» représente tout commentaire se trouvant dans la description logique qui, lui aussi, ne doit pas être pris en compte. Le terme «*(options {greedy=false;} : .)*» empêche le point de correspondre avec n'importe quel caractère jusqu'à ce qu'il rencontre un changement de ligne ou une fin de commentaire.

```
TRANSITION: 'RISE'|'FALL' ;

TIME: 'ARRIVAL'|'REQUIRED' ;

VALUE: INT | FLOAT ;

INT: ('0'..'9')+ ;

FLOAT: INT '.' INT ;

ID: FLAT | '{' BOOL '}' ;

BOOL: '0' | '1' ;

FLAT: STRING (':' STRING)* ;

STRING: ('a'..'z' | 'A'..'Z' | '_')('a'..'z' | 'A'..'Z' | '_' | '0'..'9')* ;

MULT: '*' ;
INV: '\'' ;
SEQ: '@SEQ' ;
TRI: '@TRI' ;

STMT_END: ';' ;

WS: (' '|'\t'|'\r'|'\n')+ ;

COMMENT : '//' ( options{greedy=false;}: . )* '\n' |
          '/*' ( options{greedy=false;}: . )* '*/' |
           '#' ( options{greedy=false;}: . )* '\n' ;
```

Figure 3.6 L'analyseur lexical pour la logique

3.1.2 Grammaire liée au modèle de délais

La description logique liée aux modèles de délais est différente de celle liée à la logique du circuit, car elle introduit des informations portant sur les délais, et cela, de deux façons différentes. La première est d'amener l'information nécessaire sur les modèles de délais employés par les portes logiques et l'autre est de décrire les caractéristiques pouvant faire varier les délais à l'intérieur de portes logiques provenant de bibliothèques existantes. Dans ce cas, deux extensions différentes seront utilisées : *.lmod* pour la définition d'un modèle de délai et *.llib* pour la définition de portes logiques prédéfinies. Donc, il est possible d'avoir une bibliothèque d'éléments logiques pouvant utiliser plusieurs modèles de délais définis pour une même porte, ceci étant une pratique courante dans les bibliothèques de cellules commerciales.

Il est évident que, puisque cette grammaire est basée sur celle liée à la logique, plusieurs règles font une deuxième apparition. Donc, seulement les nouvelles règles seront expliquées dans les sections suivantes. Pour une explication des autres règles, se référer à la section 3.1.1.

L'analyseur syntaxique

Selon la figure 3.7, les différents éléments composant la description logique sont classés par ordre d'importance, du haut vers le bas. Les éléments contenus dans les alvéoles sont ceux qui sont nouveaux ou qui ont changé par rapport à la grammaire liée à la logique. Cette figure est une représentation hiérarchique de l'analyseur syntaxique se trouvant à la figure 3.8.

Le premier élément qui diffère légèrement est la règle qui porte sur la fonction. Si elle correspond à une description d'une porte logique existante, le mot-clé «module» précédera l'en-tête et son contenu. Sinon, elle correspond à la définition d'un modèle de délai. Décrivons, tout d'abord, la première option qui possède les même règles concernant le contenu de la fonction. Les déclarations, par contre, sont différentes avec six possibilités différentes. Ici aussi, les équations et les instances sont prises en compte, donc les mêmes règles s'appliquent. En plus, il est possible de définir la valeur des capacités des points de contact sur une porte logique. L'unacité peut également être définie entre deux points de contact. Cette propriété sera définie en détail à la section 4.1.3, mais, pour l'instant, elle sera définie comme étant le changement d'état de la sortie en fonction du changement de l'état de l'entrée. Enfin, la définition des paramètres pour le modèle de délai utilisé, représenté par le mot-clé «delay model», suivi du type de modèle, suivi du nom du modèle pour finir avec une liste d'éléments, est présente. Cette liste est représentée par un point d'en-

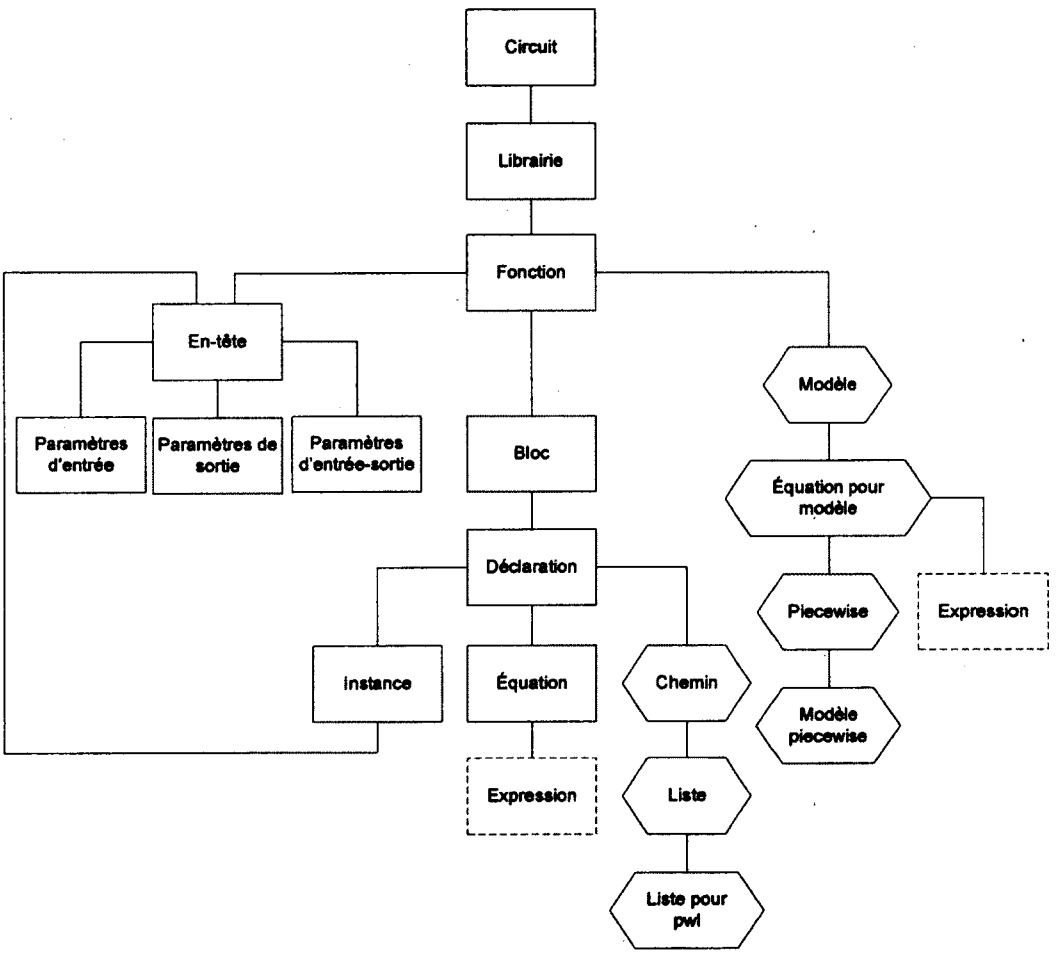


Figure 3.7 Hiérarchie des règles de l'analyseur syntaxique lié au modèle

trée, un point de sortie, la transition et une série de paramètres avec leur valeur associée. Quatre types de modèle sont supportés : linéaire, quadratique, exponentiel et linéaire par morceaux. Si le type de modèle est linéaire par morceaux, la sérié de paramètres deviendra alors une série de coordonnées. Ces éléments seront définis plus en détails dans la section 3.2.2.

Si la fonction correspond en fait à une définition d'un modèle de délai, le mot-clé «model» sera suivi du type de modèle, par exemple, linéaire, suivi du nom donné à ce modèle pour se terminer par une liste de déclarations propres au modèle. Le nom du modèle permet la description de plusieurs types de délais différents. Les déclarations du modèle peuvent être, quant à elles, représentées de quatre façons différentes.

1. Une variable égalant une expression mathématique, ce qui représente l'équation même du délai.

2. Un ensemble de coordonnées identifiant le type de modèle linéaire par morceaux. Pour l'instant, le programme ne peut supporter plus que deux coordonnées, mais la grammaire est conçue pour en prendre plusieurs.

3. Les deux autres possibilités font référence aux paramètres dans l'équation de délai et les valeurs par défaut qui leur sont attribuées. Lorsque le terme identifiant la transition (montante ou descendante) est manquant, cela signifie que la valeur est la même pour les deux.

```
circuit: library+ ;

library: ID '{' function+ '}' ;

function: 'module' user_def bloc | model ;

bloc: '{' stmt+ '}' ;

user_def: ID '(' inparam? ';' outparam ';' ioparam? ')' ;

inparam: ID (',' ID)* ;
outparam: ID (',' ID)* ;
ioparam: ID (',' ID)* ;

stmt: 'delay model' MODEL ID? '[' path+ ']' SIMT_END |
      ID ID '[' UNATENESS ']' | ID '=' expr SIMT_END |
      ID (ID '.')? user_def SIMT_END |
      ATTRIBUTE ID VALUE SIMT_END | ATTRIBUTE ID SUM SIMT_END ;

path: ID ID TRANSITION? list+ ;

list: ID VALUE | ID SUM | 'pwl' '{' ('(' pwl_list ')')+ '}' ;

pwl_list: (VALUE ',')+ VALUE ;

model: 'model' MODEL ID? '{' model_eq+ '}' ;

model_eq: ID '=' expr SIMT_END | piecewise |
          ID ':=' VALUE TRANSITION? SIMT_END | ID ':=' SUM TRANSITION? SIMT_END ;

piecewise: 'pwl' '{' '(' pwl_model ')' '}' SIMT_END ;

pwl_model: (ID ',')+ ID ;

expr: multExpr (addop multExpr)* ;

multExpr: phase (multop phase)* ;

addop: '+' | '-' ;

multop: MULT | ' ' | '/' | '**' ;

phase: atom (INV)? | '!' atom ;

atom: ID | VALUE | pre_def | '(' expr ')' ;

pre_def: SEQ '(' param_seq ')' | TRI '(' param_tri ')' ;
param_seq: ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ',' ID ;
param_tri: ID ',' ID ;
```

Figure 3.8 L'analyseur syntaxique pour le modèle

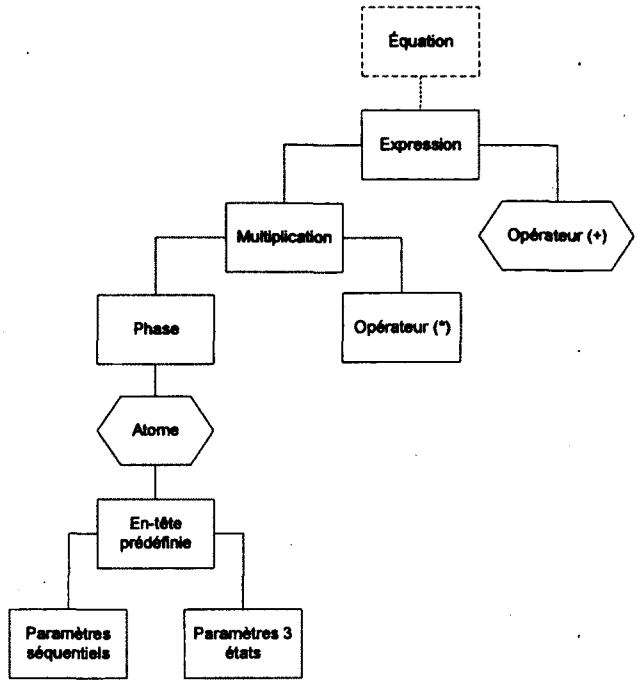


Figure 3.9 Hiérarchie des règles de l'analyseur syntaxique lié au modèle (suite)

La figure 3.9 présente la suite de la structure de l'arbre vu à la figure 3.7, qui permet le traitement des expressions de base. Ici, la seule différence avec la logique est l'utilisation d'opérateurs présents dans l'équation du modèle de délai, comme la soustraction, la division et l'exponentiation. Ces opérateurs sont utilisés spécifiquement dans les équations servant à déterminer les délais dans les portes logiques. Une deuxième différence est que la plus simple expression, identifiée par la règle «atom» peut être également définie par une valeur numérique, ce qui est utile principalement lors de l'attribution des valeurs de capacités aux points de contact de l'équation.

L'analyseur lexical

Les régles de l'analyseur lexical sont assez simples, alors seules certaines d'entre elles seront expliquées ici. La définition de l'ensemble complet de règles se trouve à la figure 3.10.

- La règle «**MODEL**» définit les types de modèles de délai pouvant être supportés par la grammaire. Ici, nous avons les types linéaire, quadratique, exponentiel et linéaire par morceaux.

- La règle «**ATTRIBUTE**» définit un attribut appartenant à un objet de la fonction, comme la capacité d'un point de contact. D'autres attributs peuvent être ajoutés à cette liste ne demandant aucune modification à la grammaire.

- La règle «**SUM**» ne fait qu'indiquer que la valeur numérique est calculable par la somme des capacités sur l'interconnexion.
- La règle «**UNATENESS**» définit l'unacité entre l'entrée et la sortie d'une équation. Elle peut être soit unate positive, unate négative ou binate.

```
MODEL: 'linear' | 'quadratic' | 'exponential' | 'piecewise' ;

ATTRIBUTE: 'CAP' ;

SUM: 'SUM' ;

TRANSITION: 'RISE' | 'FALL' ;

UNATENESS: '+UN' | '-UN' | 'BI' ;

VALUE: INT | FLOAT ;

INT: ('0'..'9')+ ;

FLOAT: INT '.' INT ;

ID: STRING | '{' BOOL '}' ;

STRING: ('a'..'z' | 'A'..'Z' | '_')('a'..'z' | 'A'..'Z' | '_' | '0'..'9')* ;

BOOL: '0' | '1' ;

EXP: '**' ;
DIV: '/' ;
MULT: '*' ;
INV: '\'' ;
SEQ: '@SEQ' ;
TRI: '@TRI' ;

STMT_END: ';' ;

WS: (' '|'\t'|'\r'|'\n')+ ;

COMMENT : '//' ( options{greedy=false;}: . )* '\n' |
          '/*' ( options{greedy=false;}: . )* '*/' |
           '#' ( options{greedy=false;}: . )* '\n' ;
```

Figure 3.10 L'analyseur lexical pour le modèle

3.2 Mise en œuvre du réseau booléen

Pour s'assurer l'accès à l'ensemble des bibliothèques par un élément unique, le modèle de conception («*design pattern*») du singleton a été utilisé. Ce modèle assure l'unicité de l'objet en plus d'y pourvoir un point d'accès global[2].

3.2.1 Logique combinatoire et séquentielle

Pour ce qui est de la logique combinatoire, la première information détectée par le générateur est le nom de la bibliothèque utilisée. Ce nom est utilisé comme clé dans une table de hachage et la valeur qui lui sera associée est un objet (lui-même un tableau de hachage) comprenant toutes les fonctions de cette bibliothèque. L'exemple illustré à la figure 3.11 démontre les principaux éléments d'une description logique. Il s'agit d'un multiplexeur à 8 entrées appartenant à la bibliothèque *Multiplexors* comprenant 1 équation de logique combinatoire et 11 instances de fonctions existantes appartenant à une autre bibliothèque. Le mot-clé *LIB* permet à l'analyseur syntaxique de détecter le nom d'une bibliothèque.

```
LIB Multiplexors{
  MUX8(a, b, c, d, e, f, g, h, sel0, sel1, sel2; out;){
    bloc1 motorolaXZ456.INV(sel0; _sel0;);
    bloc2 motorolaXZ456.INV(sel1; _sel1;);
    bloc3 motorolaXZ456.INV(sel2; _sel2;);

    bloc4 motorolaXZ456.AND4(a, _sel0, _sel1, _sel2; a1;);
    bloc5 motorolaXZ456.AND4(b, sel0, _sel1, _sel2; b1;);
    bloc6 motorolaXZ456.AND4(c, _sel0, sel1, _sel2; c1;);
    bloc7 motorolaXZ456.AND4(d, sel0, sel1, _sel2; d1;);
    bloc8 motorolaXZ456.AND4(e, _sel0, _sel1, sel2; e1;);
    bloc9 motorolaXZ456.AND4(f, sel0, _sel1, sel2; f1;);
    bloc10 motorolaXZ456.AND4(g, _sel0, sel1, sel2; g1;);
    bloc11 motorolaXZ456.AND4(h, sel0, sel1, sel2; h1;);

    out = a1 + b1 + c1 + d1 + e1 + f1 + g1 + h1;
  }
}
```

Figure 3.11 Éléments d'une description logique

La fonction logique

La fonction logique est composée de son nom suivi des paramètres qui la définissent, c'est-à-dire les entrées, les sorties et les entrées-sorties. Alors, à la lecture de cette ligne, un objet

[2]Pour plus d'informations sur les modèles de conception, se référer à [Freeman *et al.*, 2004] et à [Gamma *et al.*, 1995].

pour la fonction est créé et ces informations, en plus de la bibliothèque, y seront stockées. Cet objet sera à son tour déposé dans la table de hachage associée à la bibliothèque avec, comme clé, le nom de la fonction. La figure 3.12 illustre ce processus.

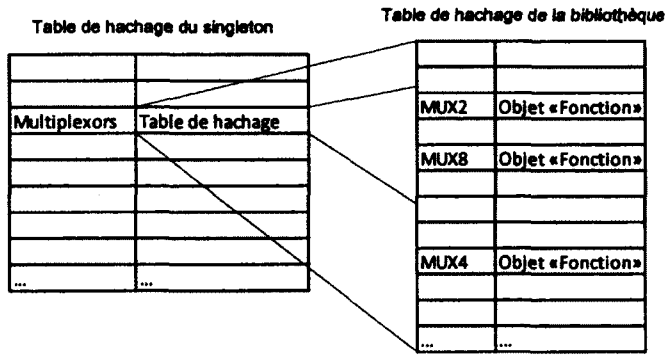


Figure 3.12 Entreposage de la bibliothèque et des fonctions

Les paramètres de la fonction, eux, subissent un traitement spécial lors de leur capture par l'analyseur syntaxique. Pour chaque type de paramètre, des connecteurs seront créés. Ceux-ci permettent de faire le pont entre les différents niveaux de hiérarchie dans le circuit, alors ils sont utilisés tant pour les fonctions que pour les instances. Le connecteur n'est pas un élément qui est observable sur un circuit, mais il est nécessaire au bon traitement des données. La raison est que chaque élément a des points de contacts sur les interconnexions et que chacun de ces points est unique. Or, ce point de contact doit se dédoubler pour se retrouver à la fois dans 2 niveaux de hiérarchie différents. Alors, le connecteur remplit cette tâche. La figure 3.13 démontre l'utilité des connecteurs dans le réseau booléen.

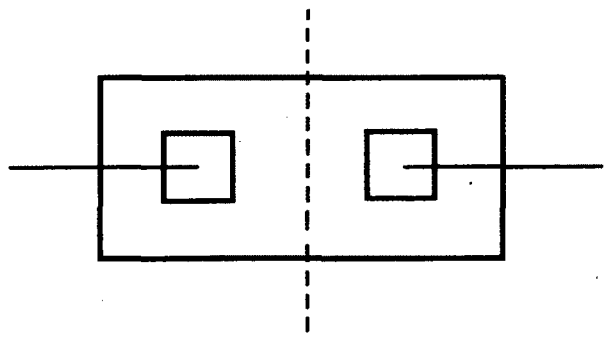


Figure 3.13 Le connecteur

Dans le modèle défini, il existe trois types de cellules : une pour la fonction, une pour l'équation et une pour l'instance. Le point de contact est utilisé dans les trois types de cellules, pour les relier au réseau booléen. Cependant, quoique le point de contact ait des caractéristiques de base identiques pour les trois types de cellules, il requiert des propriétés

propres à chacun des types de cellules. C'est pourquoi une classe abstraite a été créée afin d'être héritée par les 3 types de cellules. La figure 3.14 illustre la relation entre ces cellules.

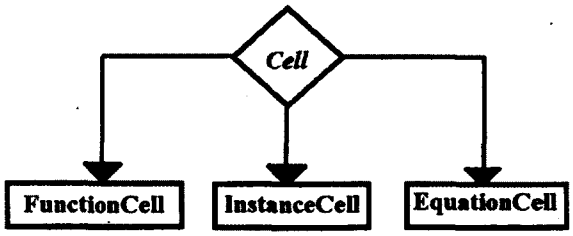


Figure 3.14 Relations entre les types de cellules

Chaque paramètre entraîne la création d'une interconnexion ayant le nom même du paramètre et celle-ci ajoute le point de contact interne du connecteur à sa liste de points de contact. L'un des éléments importants de cette structure est l'assignation d'une caractéristique indiquant si la présente fonction contient des éléments pré-existants ou non. L'intérêt de cette distinction est qu'il existe des fabriquants (par exemple, Motorola, Fujitsu, etc.) qui proposent des bibliothèques de cellules prédéfinies à leurs clients. Cependant, les concepteurs de circuits intégrés utilisent fréquemment des structures qu'ils ont eux-mêmes définies et qu'ils réutilisent fréquemment. La grammaire que nous avons établie, ainsi que le modèle interne qui en est issu, permettent ainsi cette distinction importante entre les types de bibliothèques.

L'instance d'une fonction

En revenant à la description illustrée à la figure 3.11, cette fonction contient onze instances : trois instances d'une porte **NON** et huit instances d'une porte **ET** à quatre entrées provenant d'une bibliothèque prédéfinie. Lors de la lecture de cette expression, plusieurs opérations seront exécutées, mais elles ne le seront pas toutes dans le même objet. Un objet de type instance sera créé avec comme information son nom, le nom de la fonction référenciée et la bibliothèque qui la contient. Il y a aura également, avec les paramètres présents, création de connecteurs pour chacun d'entre eux.

Les connecteurs de l'instance sont différents de ceux de la fonction, car les points de contacts n'appartiennent pas au même type de cellule. Du point de vue extérieur, l'instance n'est qu'une boîte noire et ce qu'elle contient est inconnu. Par contre, du point de vue intérieur, tous les éléments sont connus. Donc, le point de contact externe fera référence à une cellule de type instance tandis que celui interne référencera une cellule de type fonction. De plus, un pointeur vers la fonction contenant cette instance sera créé, ce qui facilitera la traversée du réseau booléen. Après la création de l'instance, les points de contact externes

doivent être ajoutés à la liste d'interconnexions de la fonction qui contient cette instance. La figure 3.15 montre la représentation d'une instance dans le réseau booléen avec tous les champs qu'elle contient.

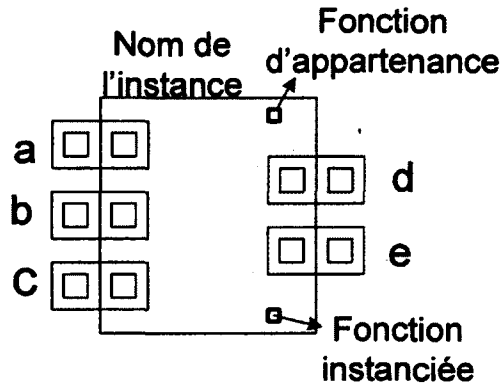


Figure 3.15 L'instance

L'instance fait partie intégrante de la fonction qui la contient. Il faut donc permettre l'accès facile à l'ensemble des instances contenues dans une fonction. À cet effet, l'objet représentant la fonction possède une liste qui énumère les instances qui y sont définies. Afin de permettre un accès rapide à cette liste, elle est implémentée sous forme de tableau de hachage où les clés sont les noms des instances de la fonction.

L'équation logique

La fonction illustrée par la figure 3.11 comprend une seule équation. Le tableau 3.1 représente les opérateurs utilisés, donc ceux pouvant être lus par l'analyseur syntaxique. La grammaire est conçue pour accepter plusieurs opérateurs différents : l'addition (+) démontre un OU logique tandis que la multiplication (*) indique un ET logique. Un ET logique sera également détecté lorsqu'il y a un espace entre deux opérandes. Le NON logique peut être déterminé par deux opérateurs différents : le point d'exclamation (!) au début de l'expression ou bien l'opérateur prime (') à la fin de l'expression. La grammaire respecte aussi la préséance des parenthèses s'il y en a. Enfin, deux opérateurs spéciaux sont supportés pour la présence d'éléments séquentiels ou de logique à trois états (*«tri-state buffers»*), @SEQ et @TRI respectivement.

La figure 3.16 montre l'objet représentant l'équation dans le réseau booléen et les champs qui lui sont associés. Le résultat et les opérandes sont des points de contact et l'arbre

Tableau 3.1 Opérateurs utilisés

Opérateur	Caractère
ET	* , « »
OU	+
NON	! , '
Logique séquentielle	@SEQ
Logique à 3 états	@TRI

contenant l'expression logique est représenté par la porte logique. Les trois autres champs correspondent à :

- Une référence à la fonction contenant cette équation afin de permettre la traversée bidirectionnelle de cette information ;
- Le modèle de délai utilisé par cette équation ;
- Une liste d'objets spécifiant l'unacité pour chaque opérande. La valeur par défaut, lors de sa création, est «binate», mais elle peut être modifiée par la suite.

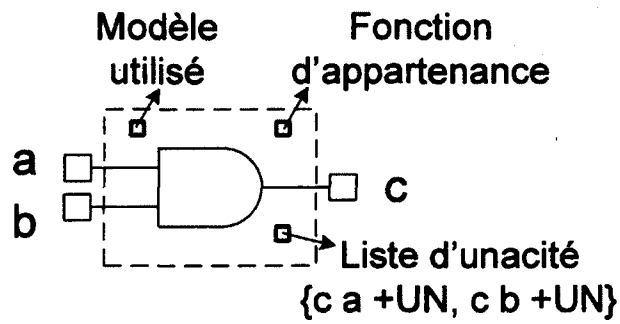


Figure 3.16 L'équation

Alors, dès que l'équation est détectée, un point de contact faisant référence à une cellule de type équation est créé pour le résultat et il est emmagasiné dans un objet de type équation. De plus, l'arbre issu de la grammaire est utilisé pour créer un arbre spécifique à l'objet équation. Cette création d'un arbre spécifique à une équation permet la traversée efficace par la suite. Donc, au fur et à mesure que cet arbre est traversé, un deuxième sera créé en parallèle qui, lui, sera facilement traversable et qui pourra être utilisé plus tard. De plus, toutes les entrées de l'équation, qui correspondent aux feuilles de l'arbre, sont identifiées et ajoutées, d'une part, à l'objet équation lors de la création de l'arbre et, d'autre part, à la liste d'interconnexions. La figure 3.17 présente le modèle abstrait d'un arbre associé à une équation logique, celle-ci étant $a + b' * c$.

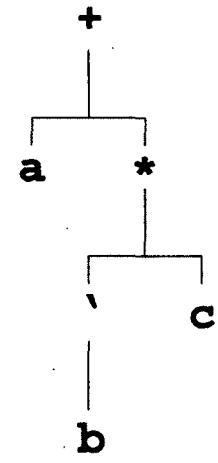


Figure 3.17 Expression logique sous forme d'arbre

En plus de créer l'objet *Équation*, il ne faut pas oublier que celui-ci appartient à une fonction. Alors, cet objet sera ensuite ajouté dans une liste prévue à cet effet appartenant à la fonction.

```
LIB flip{
  DFF(D,CK;Q,QB;){
    Q = @SEQ(D,{0},{0},{0},{0},{0},{0},CK,{0});
    QB = @SEQ(D,{0},{0},{0},{0},{0},{0},CK,{1});
  }
}
```

Figure 3.18 Description logique d'un élément séquentiel

Pour les opérateurs spéciaux, un traitement un peu particulier est approprié. Choisir un opérateur pouvant englober tous les éléments séquentiels possibles avec ces 8 entrées amène plus de simplicité. La figure 3.19 représente quatre types de portes séquentielles différentes, la bascule D, la bascule JK, la bascule SR asynchrone et la bascule T respectivement, avec les entrées permettant d'obtenir ce type de porte. Prenons l'exemple d'une bascule D, dont l'équivalent en description logique se trouve à la figure 3.18. Il est possible de remarquer les signaux d'horloge (CK) et de donnée (D) qui sont nécessaires pour les bascules D. Les autres paramètres sont, dans l'ordre, le «*Set*» Synchrone (SS), le «*Reset*» Synchrone (RS), la Priorité Synchrone (SP), le «*Set*» Asynchrone (AS), le «*Reset*» Asynchrone (AR) et la Priorité Asynchrone (AP). Ces paramètres ne sont pas utilisés par la bascule D, mais peuvent l'être pour d'autres types de bascule. Les deux paramètres de priorité indiquent quelle entrée utiliser advenant l'utilisation des deux paramètres : 0 indique que le «*Set*»

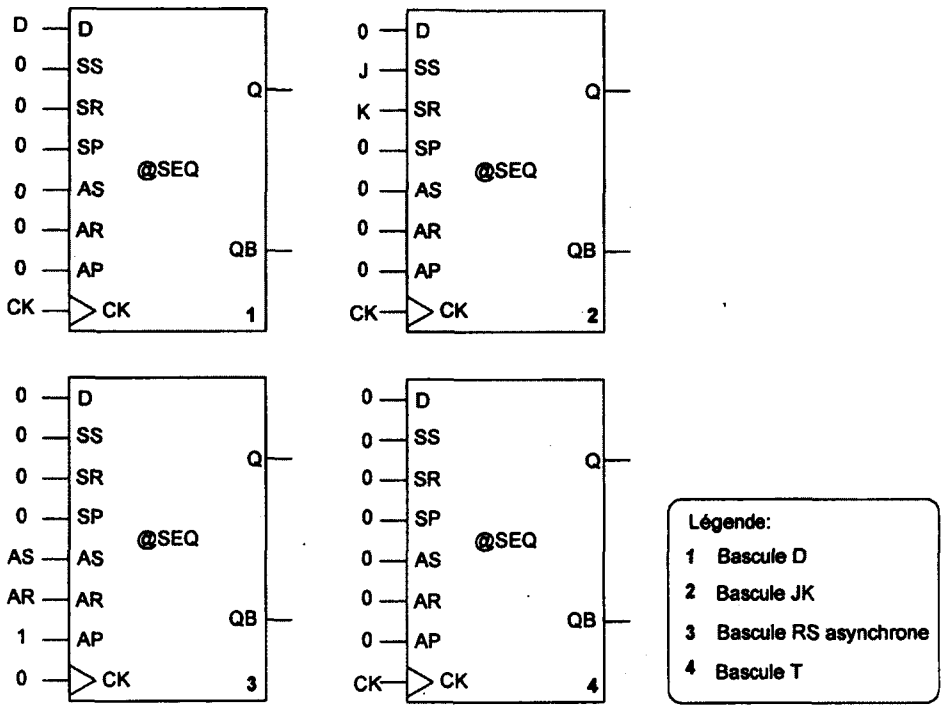


Figure 3.19 Différents type de bascules et leurs entrées respectives

est prioritaire tandis que 1 réfère au «*Reset*». Le dernier élément ne fait que mentionner la sortie, 0 étant Q et 1 étant QB.

Les éléments de logique à trois états sont, eux aussi, traités différemment des éléments de logique combinatoire. Seulement deux paramètres sont nécessaires pour déterminer la sortie : l'entrée et la décision. La décision est l'élément indiquant si le tampon est en lecture ou en écriture. À remarquer que la sortie de l'élément à trois états se trouve en «haute impédance» lorsqu'il est utilisé en lecture.

3.2.2 Logique pré-existante

La grammaire définie permet de faire la capture de la description de portes logiques existantes, par exemple produites par des compagnies offrant des services de fabrication de circuits numériques. Dans ce cas précis, en plus de la description purement logique, il est aussi important de supporter la description électrique (les caractéristiques d'opération) de la porte. Dans la section qui suit, les ajouts à la grammaire qui permettent de supporter cette fonctionnalité sont décrits. Cependant, puisqu'il y a beaucoup de choses en commun avec la section précédente, seulement les éléments nouveaux seront expliqués ici.

Le modèle

Dans tout système de synthèse, la représentation précise des délais est cruciale. Cependant, selon les circonstances, il est possible que différents modèles de délai soient appropriés. Dans le but de permettre une plus grande flexibilité, un ensemble de modèles différents pour représenter les délais sont donc proposés. L'objectif est de créer une grammaire qui permette aussi bien la description d'équations à utiliser pour faire le calcul de délais que l'utilisation de tableaux de délais mesurés, à être utilisé dans un système qui permet l'interpolation. La grammaire supporte ainsi les modèles linéaire et linéaire par morceaux, qui sont les plus fréquemment utilisés dans les bibliothèques de portes logiques commerciales. Au besoin, la grammaire peut facilement être étendue pour supporter d'autres types de modèles de délais, par exemple le modèle quadratique ou le modèle exponentiel.

Tout d'abord, les modèles de délais sont associés à une bibliothèque existante. Ainsi, toutes les portes logiques dans une bibliothèque donnée devront utiliser nécéssairement l'un des modèles présents. Plusieurs modèles de délais peuvent coexister dans une même bibliothèque. Typiquement, dans une bibliothèque commerciale, les portes logiques sont déclinées en familles de performance. Par exemple, on pourrait retrouver pour chaque porte logique quatre implémentations possibles, allant de la porte la plus rapide (qui consomme le plus de puissance) à la porte la plus petite (qui consomme peu mais qui est lente). Dans la bibliothèque correspondante, il y aurait un ensemble de modèles de délais, chacun étant utilisé pour caractériser les performance d'une famille de portes logiques. Selon la figure 3.20, le modèle linéaire se retrouvera dans la bibliothèque fictive *motorolaXZ456*. Ensuite, l'équation utilisée pour trouver le délai est indiquée. Ici, l'équation 3.1 ;

$$t = t_0 + \beta * \gamma \tag{3.1}$$

où t_0 représente le délai intrinsèque, β représente la résistance interne et γ représente la charge à la sortie, représente une équation de délai pour le modèle linéaire. Cette équation est stockée sous la même forme qu'une équation logique étant dans une fonction logique. Par la suite, nous avons les valeurs par défaut de ces paramètres, en montée et en descente. On peut remarquer qu'il n'y a qu'une seule valeur pour γ et que celle-ci n'est pas numérique. Puisque la somme des capacités à la sortie nous donne la charge sur celle-ci et que le fait qu'on soit en montée ou en descente n'influence pas ces valeurs, nous avons utilisé le mot-clé *SUM* de façon à pouvoir calculer cette charge lors du calcul des délais.

Le modèle linéaire par morceaux, lui, n'est pas représenté par une équation mathématique comme le montre la description à la figure 3.21. Ce modèle est représenté par un tableau

```
motorolaXZ456{
  model linear{
    t = t0 + beta * gamma;
    t0 := 30 RISE;
    t0 := 25 FALL;
    beta := 25 RISE;
    beta := 20 FALL;
    gamma := SUM;
  }
}
```

Figure 3.20 Exemple d'un modèle linéaire

de données comportant n dimensions. Ici, le mot-clé *pwl* indique l'utilisation du modèle linéaire par morceaux et le délai dépendra de la valeur de la charge à la sortie. Ici, encore une fois, γ est donné par la somme des capacités à la sortie. Pour limiter la complexité, un modèle en deux dimensions a été suffisant, mais la possibilité de traiter des modèles utilisant plusieurs paramètres existe.

```
motorolaXZ456{
  model piecewise{
    pwl{(gamma, t)};
    gamma := SUM;
  }
}
```

Figure 3.21 Exemple d'un modèle linéaire par morceaux

Une fois la définition des modèles établie, il ne reste qu'à définir les portes logiques qui les utiliseront. Un fait très important à noter est que les modèles sont stockés dans les bibliothèques de cellules, celles-ci étant elles-mêmes stockées dans un objet unique (un singleton) qui contient l'ensemble des bibliothèques de cellules. Cet objet singleton contenant les bibliothèques est analogue à celui qui contient l'ensemble des circuits logiques. De façon analogue, ces deux objets sont implémentés par des tableaux de hachage.

La porte logique

La structure des portes logiques prédéfinies diffère de celle des fonctions logiques vues précédemment : certains nouveaux paramètres doivent être pris en compte comme la capacité sur les points de contact. La figure 3.22 représente la description d'une porte **OU** à 2 entrées. Le mot-clé *module* indique que la fonction est une porte logique venant d'une bibliothèque déjà existante. Ici, il est important de noter que les portes logiques se voient assigner une propriété qui indique qu'elles sont différentes des fonctions logiques. Par

convention, les paramètres de la porte sont définis en majuscules et des points de contact et des interconnexions sont créés lors de la lecture par l'analyseur syntaxique.

```
module OR2(A,B;C;){
  C = A + B;
  A C [+UN]
  B C [+UN]
  delay model linear [ C A RISE t0 25 beta 12 gamma SUM
                       C B RISE t0 30 beta 15 gamma SUM
                       C A FALL t0 22 beta 20 gamma SUM
                       C B FALL t0 20 beta 25 gamma SUM];
  CAP A 63.9;
  CAP B 44.5;
  CAP C 10.2;
}
```

Figure 3.22 Description logique d'une porte existante

Le premier élément de la porte est la relation logique devant être représentée. Celui-ci est emmagasiné dans un tableau sous la forme d'un objet de type équation et ses points de contact sont ajoutés aux interconnexions ayant le même nom. Le prochain élément est le caractère d'unacité de la porte qui indique une certain relation entre la sortie et une entrée de la porte. Cette propriété est définie plus en détails dans la section 4.1.3. Cette propriété est stockée dans un objet spécialement conçu à cet effet avec l'équation.

L'information suivante est reliée au type de modèle utilisé par la porte. L'information entre crochets indique les paramètres à utiliser pour l'équation, et ce, pour chaque chemin possible entre une entrée et une sortie. Le chemin est défini avec la sortie, l'entrée et la direction du signal (en montée ou en descente). Ces paramètres sont facultatifs, car une omission entraîne automatiquement l'activation du paramètre par défaut situé dans le modèle. Enfin, la dernière information est celle des valeurs capacitives sur les points de contact de l'équation. L'objet représentant le point de contact est annoté avec l'information relative à leurs capacités.

La figure 3.24 montre le graphe de la courbe de délai pour le module décrit par la description illustrée à la figure 3.23, module utilisant un modèle linéaire par morceaux. La valeur qui sera calculée pour la charge à la sortie déterminera quelle portion de la courbe sera utilisée pour calculer le délai de la porte logique.

```
module NAND2(A,B;C;){
  C = !(A * B);
  A C [-UN]
  B C [-UN]
  delay model piecewise
          [C A RISE pwl{(10, 1)(15, 2)(25, 3)(45, 4)(85, 5)}
           C B RISE pwl{(10, 1)(15, 2)(25, 3)(45, 4)(85, 5)}
           C A FALL pwl{(10, 1)(15, 2)(25, 3)(45, 4)(85, 5)}
           C B FALL pwl{(10, 1)(15, 2)(25, 3)(45, 4)(85, 5)}];
  CAP A 4.53;
  CAP B 3.41;
  CAP C 1.58;
}
```

Figure 3.23 Description d'une porte avec le modèle linéaire par morceaux

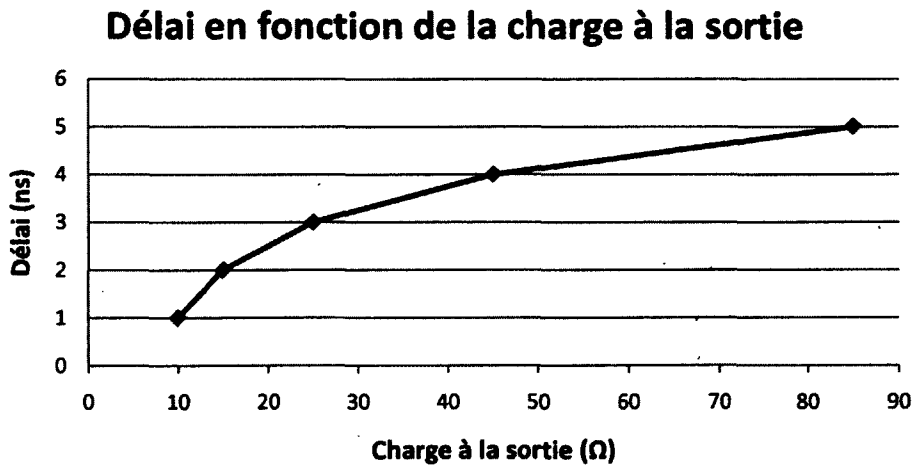


Figure 3.24 Graphe utilisant le modèle linéaire par morceaux

3.3 Opérations utilisées par la synthèse logique

Une fois la lecture de la description logique terminée, la représentation interne du circuit est complétée. Alors, il est désormais possible d'effectuer des opérations sur celui-ci. Cependant, avant de pouvoir utiliser des fonctions provenant de la synthèse logique, comme l'aplanissement et le groupement, il faut être capables de copier adéquatement des fonctions. En effet, lors des opérations de synthèse, il est fréquent que des portions de circuits doivent être dupliquées pour permettre la modification sélective de la structure logique. Par exemple, dans un additionneur 32 bits consistant en 32 instances d'un additionneur 1 bit, il pourrait être avantageux de modifier certaines des instances pour accélérer les derniers étages de l'additionneur. Pour ce faire, le simple processus de copie ne permet pas d'obtenir les résultats escomptés, car il crée un nouvel objet, mais celui-ci détient toujours

les références du premier objet. Alors, des modifications sur un objet entraînent également des modifications sur l'autre objet. C'est pourquoi le processus de clonage a été incorporé au programme.

3.3.1 Le processus de clonage

Le clonage permet de faire une copie d'un objet en profondeur ("*deep*") plutôt qu'en surface ("*shallow*"). En d'autres termes, il permet de ne pas simplement faire une copie de l'objet principal, mais aussi de tous les objets en faisant partie, et ce, de façon hiérarchique. La figure 3.25 montre la différence entre ces deux copies. Les éléments en pointillés représentent les éléments qui sont différents après l'opération de clonage. Le premier clone est une fonction différente de l'originale, mais les éléments qui en font partie sont les mêmes. Le deuxième clone, lui, est une fonction différente de l'originale et les éléments qui en font partie sont également différents puisque la copie s'est faite sur chacun de ces éléments.

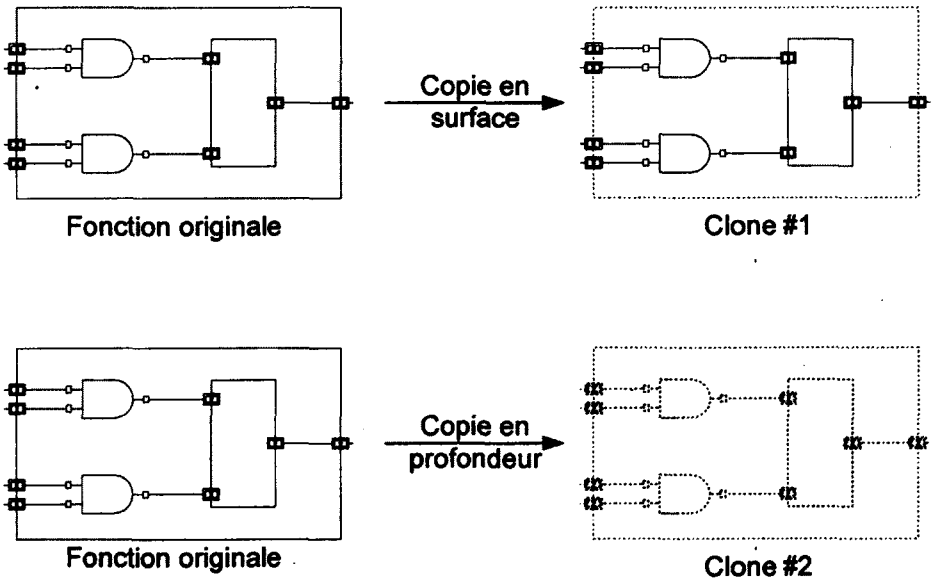


Figure 3.25 Les deux types de clonage

La raison pour laquelle ce processus est nécessaire est qu'on doit permettre la modification de fonctions copiées de façon indépendante. Certaines opérations demandent la modification d'une copie de la fonction sans altération de l'original ce qui implique l'utilisation du clonage. De plus, tous les objets à l'intérieur de la fonction, que ce soit une interconnexion, un point de contact ou un autre objet, sont très fortement reliés. Par exemple, supposons que cinq points de contact se retrouvent sur une interconnexion : l'objet représentant cette dernière doit aussi être lié aux objets représentant les points de contact. De nombreux ob-

jets sont ainsi doublement liés, pour permettre l'accès rapide à l'ensemble de l'information. Alors, il faut s'assurer du respect de ces relations.

Table de hachage pour le clonage

Equation1	Clone de Equation1
Instance1	Clone de Instance1
Equation2	Clone de Equation2
Point de contact b2	Clone de Point de contact b2
Point de contact a1	Clone de Point de contact a1
Interconnexion a	Clone de Interconnexion a
...	...

Figure 3.26 La table de hachage pour le clonage

Pour y arriver, deux conditions doivent être remplies. La première est la création d'une table de hachage, illustrée à la figure 3.26, pouvant accepter tous les éléments possibles et la deuxième est l'implémentation de l'interface *Cloneable* par toutes les classes concernées. La table de hachage va se remplir au fur et à mesure que les champs à l'intérieur de la fonction sont lus. La clé est l'objet lui-même tandis que la valeur est son clone. Une vérification est toujours effectuée pour vérifier si l'objet sur le point d'être cloné est présent dans la table. Lorsqu'un objet n'étant pas dans la table est détecté, un clone de cet objet est créé, d'où l'importance de la deuxième condition. Les opérations d'aplanissement et de groupement utilisent le clonage parce qu'elles modifient les fonctions.

3.3.2 La fonction d'aplanissement

Cette opération est capitale pour permettre aux opérations de synthèse logique d'opérer efficacement. En effet, l'aplanissement sélectif permet de regrouper des portes logiques apparaissant au départ dans des niveaux de hiérarchie différents, et ainsi d'atteindre une meilleure qualité d'optimisation. La figure 3.27 nous montre l'état d'une fonction avant et après l'opération d'aplanissement. Cette opération est quand même assez flexible, car elle prend en entrée le nombre de niveaux devant être ramenés au niveau supérieur et le nom de la nouvelle fonction crée.

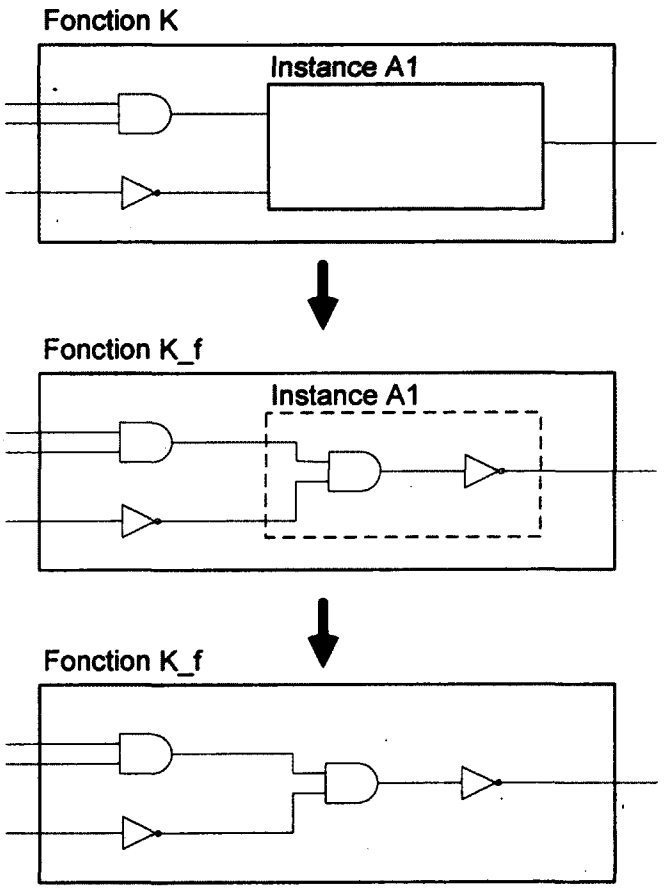


Figure 3.27 Opération d'aplanissement

Alors, la fonction K contient deux équations logiques, un inverseur et une porte **ET**, et une instance *A1*. Lors de l'aplanissement, la fonction principale est clonée puisque des modifications vont lui être apportées. Par la suite, le contenu des instances est connu pour ensuite prendre ces éléments et les rajouter à la fonction principale. Finalement, on obtient la fonction clonée K_f contenant quatre équations logiques et l'instance *A1* qui n'existe plus.

L'algorithme présenté à la figure 3.28 permet de mieux comprendre la fonction d'aplanissement. Les paramètres d'entrée sont, en plus de la profondeur et du nom de la nouvelle fonction, un paramètre indiquant si c'est la première fois qu'on exécute la méthode puisque celle-ci est récursive et un autre qui est la fonction aplanie définie plus tard. Donc, ce paramètre n'est pas utilisé lors de la première passe. La première étape est de déterminer s'il s'agit, en fait, d'une première passe et, si c'est le cas, une nouvelle fonction est créée en clonant la fonction originale et elle est stockée dans une bibliothèque spécialement conçue pour les fonctions aplanies.

```
flatten(profondeur, premier niveau, input, fonction aplanie)

  IF(première passe == VRAI)

    clone = reference.clone();
    clone.setName(input);
    logicRepository.add(new library());
    library.add(clone);

  FOR(chaque instance)

    cloneInstance = instance.getFonction().clone();

    FOR(chaque interconnexion)

      IF (clone.netList.contains(cloneInstance.net.name))
        interconnexion.merge(clone.interconnexion);

      ELSE
        cloneInstance.interconnexion.changeName();
        clone.netList.add(cloneInstance.interconnexion);

    clone.addAllEquation(cloneInstance.getEquationArray);
    clone.addAllInstance(cloneInstance.getInstanceArray);

  clone.remove(chaque instance);
  checkInstances(clone);
  flatten(profondeur - 1, FAUX, clone);
```

Figure 3.28 Algorithme pour aplanir une fonction

Ensuite, pour chaque instance se trouvant dans la fonction, un processus d'aplanissement est effectué. Tout d'abord, en allant chercher la fonction référencée par l'instance, les interconnexions devant être fusionnées avec les interconnexions au niveau supérieur sont déterminées. Ce processus, bien qu'en apparence complexe, est simple à comprendre et est illustré à la figure 3.29. Les points de contacts sur l'interconexion interne, sauf celui sur la cellule, sont transférées sur l'interconnexion supérieure qui, elle, se voit retirer le point de contact sur l'instance. Les autres interconnexions présentes dans l'instance, celles étant uniquement à l'interne, se voient rajoutées à la liste d'interconnexions de la fonction aplanie avec un nouveau nom reflétant le fait qu'elles proviennent d'une instance aplanie. Cette mesure est mise en place pour éviter d'avoir deux interconnexions ayant le même nom. Si on se réfère à l'illustration, l'interconnexion x de l'instance voit son nom changer pour *A1 :A :x* pour éviter l'éventuel conflit.

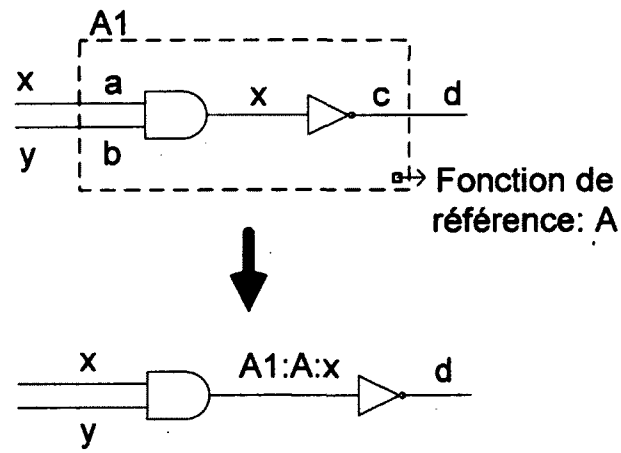


Figure 3.29 Processus de fusion de l'interconnexion

Enfin, toutes les instances et équations se trouvant dans l'instances sont ajoutées à leur liste correspondante dans la fonction aplanie. Par contre, l'instance venant d'être aplanie doit être enlevée de sa liste puisqu'elle ne s'y trouve plus. Finalement, si la fonction doit être aplanie encore d'un niveau, la méthode s'appelle récursivement jusqu'à ce que le nombre de niveaux soit atteint.

3.3.3 La fonction de groupement

La fonction de groupement permet de grouper plusieurs objets de la fonction, telles que instances et équations, pour en faire une fonction pouvant être réutilisée par la suite dans d'autres fonctions sous la forme d'instance. Cette opération est en quelque sorte l'inverse de l'opération d'aplanissement décrite auparavant. La figure 3.30 nous montre une fonction avant et après l'opération de groupement.

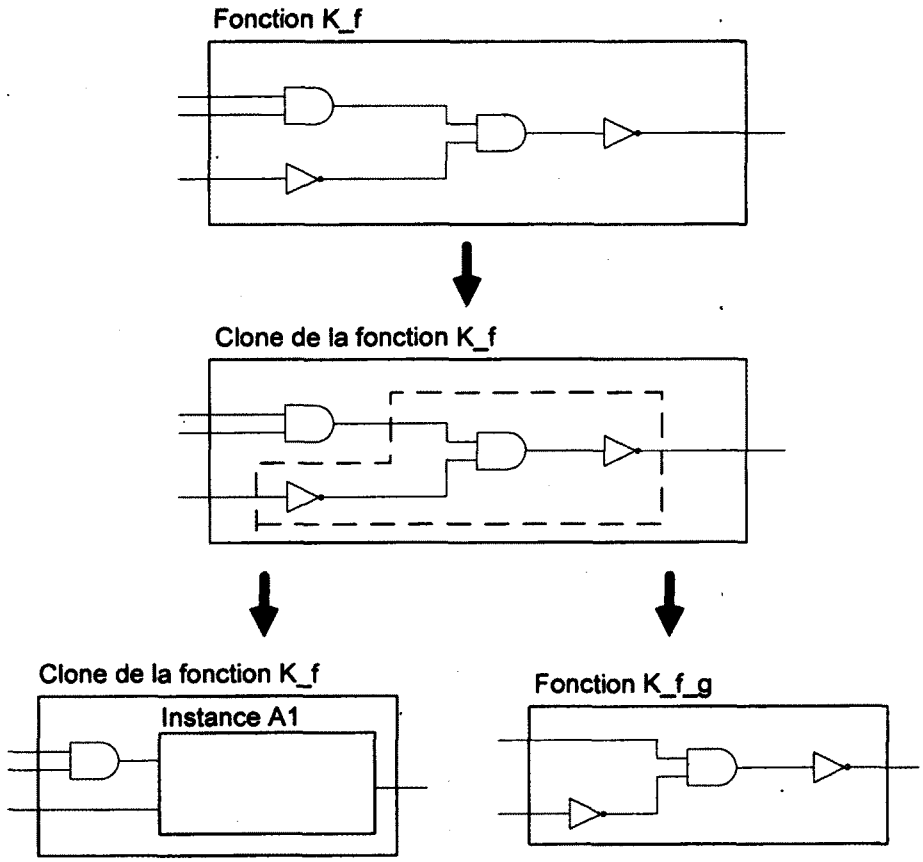


Figure 3.30 Opération de groupement

Nous pouvoir voir, ici, que la fonction utilisée est la même que celle obtenue après l'opération d'aplanissement. Alors, la première opération consiste à faire un clone de cette fonction, mais celle-ci peut garder le même nom puisqu'elle est stockée dans une autre bibliothèque. Un choix est fait à savoir quels éléments de la fonction doivent être groupés. Ensuite, une séparation des interconnexions s'effectue pour séparer les éléments groupés de la fonction originale. Enfin, deux nouvelles fonctions sont créées : l'une contenant les éléments groupés et l'autre contenant l'instance de cette fonction.

L'algorithme présenté à la figure 3.31 permet de mieux comprendre la fonction de groupement. Tout d'abord, les paramètres d'entrée sont le nom de la nouvelle fonction créée et deux listes regroupant les équations et les instances qui seront groupées ensemble. Puisque la nouvelle fonction créée est basée sur une fonction existante, celle-ci doit être cloner pour éviter de modifier les informations s'y trouvant. Quant à la fonction qui sera instanciée, un nouvel objet est créé qui sera rempli, par la suite, avec les équations, instances et interconnexions nécessaires. Ces deux fonctions, le clone et la fonction à être instanciée, seront ensuite placées dans une bibliothèque spécialement conçue pour le groupement. La fonction résultant du groupement devient également une instance dans la fonction clonée et est ainsi rajoutée à la liste.

Par la suite, tous les points de contacts qui seront affectés par le groupement sont déterminés et ceux-ci sont placés dans une liste. Ensuite, la séparation de chaque interconnexion dans le clone doit être déterminée lors du groupement. Il est bien de noter également que si l'interconnexion se retrouve entièrement affectée par le groupement, elle sera placée dans une liste temporaire de façon à être déplacée une fois la vérification de toutes les interconnexions terminée.

Le processus de division de l'interconnexion est un peu plus élaboré que celui de la fusion. La création des connecteurs qui seront placés sur la division de l'interconnexion est effectuée, un pour l'instance et un pour la fonction instanciée, et l'un des points de contact de chacun de ces connecteurs est ajouté à la liste des points de contact appartenant à l'interconnexion. Ensuite, une comparaison de tous les points de contact dans cette liste avec la liste contenant tous les points de contact affectés par le groupement est effectuée. Lorsque la comparaison est vraie, alors le point de contact se déplace d'une interconnexion à l'autre.

Une fois la liste des interconnexions traversée, nous pouvons déplacer celles qui font entièrement partie de la nouvelle fonction. Ensuite, pour s'assurer que les points de contact sont bien nommés, une mise à jour des informations concernant la cellule est effectuée. Il ne reste plus qu'à déplacer les instances et les équations devant être groupées dans la nouvelle fonction et de mettre les informations concernant chaque cellule d'instance dans la fonction nouvellement créée.

```
group(input, equation[], instance[])

  clone = fonction.clone();
  cloneGroup = new Fonction();
  cloneGroup.setName(input);

  logicRepository.add(new library());
  library.add(cloneGroup);
  library.add(clone);

  clone.addInstance(new Instance());
  allPins = getAllPins();

  FOR (chaque interconnexion)

    split = checkSplitting();

    IF(split == VRAI)

      IF(allPins.contains(tous les points de contact))
        temp.add(clone.interconnexion);
      ELSE
        clone.interconnexion.split(clone, cloneGroup, allPins);

  IF(temp.isNotEmpty())
    clone.netList.move(cloneGroup, temp.getAll());

  updateElements();

  IF(equation[].isNotEmpty())

    FOR(chaque équation)
      cloneGroup.add(equation);
      clone.remove(equation);

  IF(instance[].isNotEmpty())

    FOR(chaque instance)
      cloneGroup.add(instance);
      clone.remove(instance);

  checkInstances(cloneGroup);
```

Figure 3.31 Algorithme pour grouper des éléments d'une fonction

CHAPITRE 4

GESTION DES BUDGETS DE DÉLAI

4.1 Préparation à la gestion des budgets de délais

Comme vu précédemment au chapitre 2, pour obtenir un budget de délai sur un bloc donné, les temps d'arrivée et les temps requis sur ce bloc doivent être connus. Il est donc important que l'infrastructure de représentation de circuits logiques soit complémentée par un système de calcul et de gestion des délais. Ce système de calcul et de gestion des budgets de délai est un module qui s'attache à la plate-forme principale de synthèse. Dans ce qui suit, l'infrastructure développée pour supporter et calculer les délais dans les circuits logiques sera d'abord présentée. Par la suite, la façon dont cette infrastructure est utilisée pour permettre le calcul de budgets de délais sera observée.

4.1.1 Conception de la structure

Tout d'abord, afin de ne pas modifier les valeurs présentes dans les réseaux booléens originaux, le calcul de la propagation des temps d'arrivée et des temps requis se fera à l'intérieur d'un environnement de test spécialement conçu à cet effet. Puisque posséder plusieurs instances de cet environnement de test n'est pas désirable, il est créé en même temps que l'objet englobant les bibliothèques en utilisant le modèle de conception du singleton. Pour permettre plusieurs tests différents, cet environnement comprend une table de hachage ayant comme clé un nom unique et comme valeur l'objet sur lequel la gestion des délais va s'exécuter.

Cet objet est une instance de la fonction globale à tester, annotée avec les contraintes temporelles spécifiques au test appliqué. L'information s'y trouvant est le nom unique de l'objet, le nom de la fonction et la bibliothèque dans laquelle elle se trouve, un objet contenant les contraintes globales pour la fonction testée, une liste pour emmagasiner les chemins critiques déterminés par le test ainsi qu'une table d'annotation, qui est simplement une table de hachage. Il est donc possible d'effectuer plusieurs tests différents afin de déterminer, à chaque fois, les chemins critiques obtenus.

La table d'annotation, présentée à la figure 4.1, est l'élément vital relié au stockage de l'information sur les délais. Chaque clé de cette table représente un objet ayant besoin

d'être annoté se trouvant dans la fonction lors du calcul des délais qui sont les points de contacts, les interconnexions, les équations et les instances de fonctions. Les valeurs associées à ces objets varient également en fonction de l'information qui est requise. Ainsi, les interconnexions, les équations et les instances sont annotées avec un objet générique permettant l'ajout d'informations arbitraires. Ce mécanisme pourrait par exemple être utilisé pour représenter la longueur des interconnexions, le modèle de délai utilisé, etc.

Les points de contact, eux, pointent vers un objet contenant toute l'information concernant les temps d'arrivée et les temps requis, en montée et en descente, à cet endroit. Quant aux instances de fonction, elles pointent vers une autre table d'annotation. Cette nouvelle table de hachage est présente seulement à cet endroit et contient toute l'information sur la fonction instantiée. De cette façon, le problème de hiérarchisation est facilement résolu, car, peu importe le nombre de niveaux présents dans la fonction, tous ses éléments y seront présents.

Plusieurs éléments sont primordiaux à la propagation des temps d'arrivée et des temps requis. Le traitement de cette information doit se faire de façon à rendre le système plus robuste sans pour autant le compliquer davantage. La solution apportée est représentée à la figure 4.1.

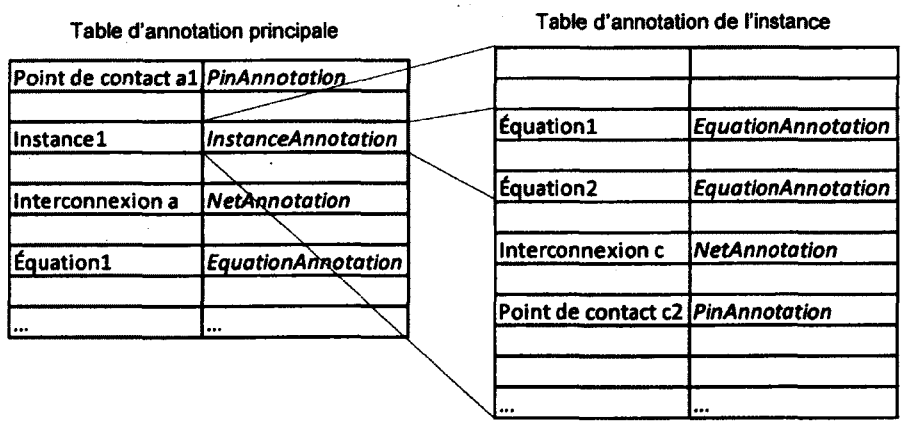


Figure 4.1 La table d'annotation

Une table d'annotation est attachée à la fonction sur laquelle les temps sont propagés. Cette table contient tous les éléments de la fonction pouvant être modifiés lors de cette opération. De plus, la table d'annotation liée à l'instance permet une hiérarchisation des multiples niveaux pouvant être présents dans le circuit.

Avant le calcul des temps d'arrivée et des temps requis, cette table d'annotation doit contenir tous les éléments se trouvant dans la fonction. Alors, chaque interconnexion ainsi

que tous les points de contacts s'y trouvant et chaque équation se voient ajoutés à la table. De plus, chaque instance présente possède aussi sa propre table d'annotation, qui contient le même type d'information, mais pour les éléments contenus dans l'instance. Pour compléter le système de calcul et de gestion des délais, il faut aussi permettre l'ajout de contraintes temporelles. C'est ce qui est présenté dans la prochaine section.

4.1.2 Établissement des contraintes

L'établissement des contraintes permet deux choses : la première est de spécifier l'environnement dans lequel le circuit existe, la deuxième est de pouvoir spécifier l'environnement pour chaque niveau hiérarchique existant dans le circuit. Étant donné le nombre élevé de contraintes existantes pouvant être appliquées sur un circuit, le système est limité, en plus des contraintes temporelles, aux capacités sur les sorties et aux résistances sur les entrées. Les autres contraintes, bien qu'importantes, ne sont pas utiles pour la gestion des budgets de délais. Cependant, le mécanisme en place permet de facilement ajouter et gérer toute contrainte additionnelle jugée nécessaire (par exemple, la puissance maximale, les vecteurs de test, l'inductance, etc.).

La différence entre les contraintes globales et les contraintes internes est que les contraintes globales doivent être déterminées. Donc, pour ce qui est des contraintes globales, les capacités pour chaque sortie et les résistances pour chaque entrée sur le circuit sont déterminées par l'usager. Les contraintes temporelles, elles, sont déjà établies lors de la lecture de la description logique. Un champ spécial est utilisé pour définir les temps d'arrivée et les temps requis du circuit, et ce, en montée et en descente. La méthode qui pose les contraintes sur le circuit prend alors chaque valeur temporelle, stockée sur les points de contacts périphériques, et les ajoute dans la table d'annotation.

Pour les contraintes internes, les valeurs des capacités et des résistances sont définies par le circuit et les bibliothèques de cellules utilisées. Pour le moment, la somme des capacités sur une interconnexion est ce qui détermine la contrainte d'opération de l'interconnexion. Dans le futur, le système d'annotation existant pourrait être utilisé pour ajouter de l'information plus complexe, comme la résistance linéaire, l'inductance mutuelle, etc. La somme des capacités sur une interconnexion devient la contrainte pour celle-ci et la résistance est déterminée par la porte logique précédent l'instance. Les contraintes temporelles, elles, sont définies lors de la propagation des temps d'arrivée (section 4.2.1) et des temps requis (section 4.2.2).

La somme des capacités

L'algorithme pour calculer la somme des capacités sur une interconnexion doit être en mesure de pénétrer dans les instances s'y trouvant et d'en ressortir une fois le travail accompli. En effet, il arrive fréquemment dans un circuit numérique qu'une interconnexion traverse plusieurs niveaux hiérarchiques de la descripton. La figure 4.2 montre l'algorithme qui calcule cette valeur et qui la retourne pour qu'elle soit utilisée.

```
getTotalCap(annotationTable, instanceArray, tagTable)

  somme = this.capacitance;
  annotation = getInstanceAnnotation(annotationTable, instanceArray);

  FOR(chaque point de contact sur interconnexion)

    IF(point de contact ne se trouve pas dans la tagTable)
      tagTable.put(point de contact);

      IF((point de contact).getType() == CHARGE ||
        (point de contact).getType() == INOUT)

        IF((point de contact).getCell() instanceof FunctionCell)

          position = (point de contact).getPosition();

          IF(instanceArray.isEmpty() == FAUX)
            somme = somme + annotation.getCapFromConstraints(position);
          ELSE
            somme = somme + TestInstance.getCapFromConstraints(position);

        ELSE IF((point de contact).getCell() instanceof InstanceCell)

          masterPin = getAssociatedPin(point de contact);
          somme = somme + masterPin.getNet().getTotalCap();

        ELSE
          somme = somme + (point de contact).getCapacitance();

      ELSE
        somme = somme + (point de contact).getCapacitance();
    ELSE
      CONTINUE;

  RETURN somme;
```

Figure 4.2 Algorithme pour le calcul de la somme des capacités

Les paramètres d'entrée sont la table d'annotation, un tableau d'instance et une table d'étiquettes. La table d'annotation n'est utile que pour le cas où le point de contact, sur

l'interconnexion, est un paramètre de sortie ou d'entrée/sortie d'une instance se trouvant dans la fonction globale. En effet, lors de l'établissement des contraintes de l'instance, la somme des capacités a déjà été calculée et stockée dans l'objet de type *Contraintes* appartenant à l'annotation sur l'instance se trouvant dans la table d'annotation. Le tableau d'instances contient la liste des instances qui ont été traversées en termes de profondeur et n'est utilisé qu'avec la table d'annotation. Quant à la table d'étiquettes, elle permet de s'assurer qu'on ne passe pas deux fois par le même point de contact et, ainsi, évite la possibilité de faire une boucle sans fin.

Tout d'abord, on égale la somme à la valeur de la capacité parasite de l'interconnexion. Ensuite, pour chaque point de contact sur l'interconnexion, on l'ajoute à la liste d'étiquettes s'il n'y est pas déjà et, dépendamment du type de point de contact et de la cellule associée, les opérations varient. Pour un point de contact qui est en charge, c'est-à-dire qui est l'entrée d'une équation ou bien le point de contact « gauche » d'un connecteur, et pour le point de contact pour les entrées/sorties, trois possibilités s'offrent à nous.

1. Lorsqu'il s'agit d'une équation, il faut ajouter la capacité du point de contact à la somme.

2. S'il s'agit d'une fonction, il faut prendre la valeur stockée dans la contrainte, qui peut être celle d'une instance ou de la fonction globale et l'ajouter à la somme.

3. Enfin, s'il s'agit d'une instance, il faut aller chercher la fonction de référence et déterminer le point de contact lié au point de contact sur l'instance avant de pouvoir appeler la fonction de façon récursive.

Par contre, si le point de contact est en conduction, on ne fait que rajouter la capacité du point de contact à la somme. La figure 4.3 montre toutes les possibilités de calculs sur une interconnexion. Le point de contact de gauche se trouve à l'intérieur de deux instances ce qui implique que le calcul des contraintes a été calculé deux fois préalablement. Cependant, au niveau supérieur, la recherche de points de contact s'effectue en plongeant dans toutes les instances rencontrées jusqu'à ce que des points de contact appartenant à des équations logiques soient trouvés. Ici, trois points de contacts ont été trouvés, soit un provenant d'un niveau de hiérarchie différent.

Détermination de la résistance d'entrée

La figure 4.4 représente un algorithme pour déterminer la résistance d'entrée et celui-ci est semblable à l'algorithme montré à la figure 4.2, puisque la structure est similaire. Cependant, il comporte quelques différences devant être notées. Sur l'interconnexion, il

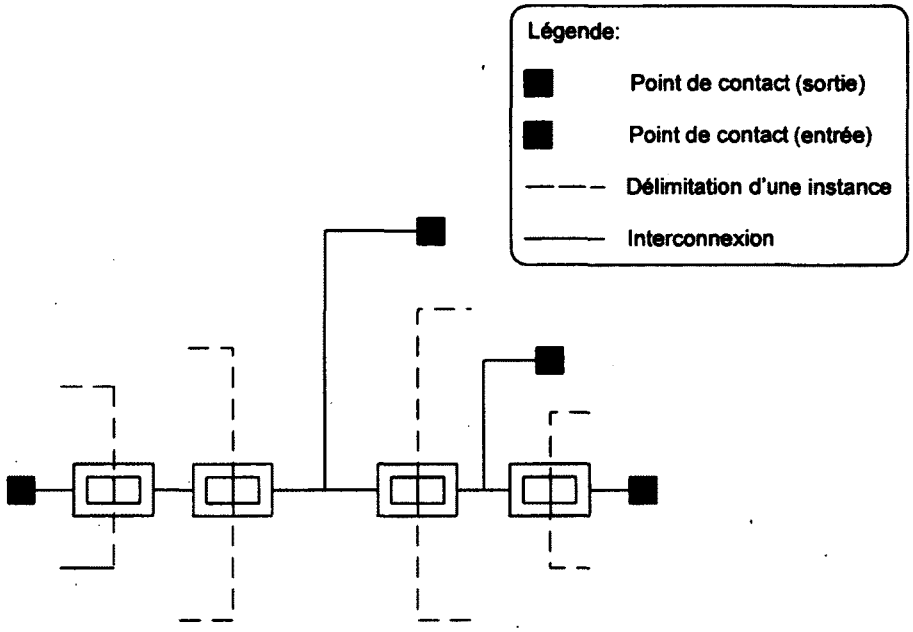


Figure 4.3 Exemple de somme des capacités sur une interconnexion

n'y a qu'une seule porte dont la sortie est branchée à l'interconnexion et nous voulons la résistance qui est associée au point de contact de cette sortie. Alors, tout comme l'algorithme précédent, il faut aller chercher l'information de façon hiérarchique, mais des étapes supplémentaires sont requises. La résistance d'entrée se trouve dans le modèle associé à l'équation. Si l'équation provient d'une bibliothèque existante, nous prenons la valeur fournie par le modèle. Sinon, la valeur choisie proviendra de la liste d'attributs associée au modèle de l'équation. Si jamais la valeur de la résistance était nulle, alors la valeur par défaut serait choisie.

```
getInputDrive(annotationTable, instanceArray, tagTable)

  drive = this.resistance;
  annotation = getInstanceAnnotation(annotationTable, instanceArray);

  FOR(chaque point de contact sur interconnexion)

    IF(point de contact ne se trouve pas dans la tagTable)
      tagTable.put(point de contact);

      IF((point de contact).getType() == DRIVE ||
        (point de contact).getType() == INOUT)

        IF((point de contact).getCell() instanceof FunctionCell)
          position = (point de contact).getPosition();

          IF(instanceArray.isEmpty() == FAUX)
            drive = drive + annotation.getCapFromConstraints(position);
          ELSE
            drive = drive + TestInstance.getCapFromConstraints(position);

        ELSE IF((point de contact).getCell() instanceof InstanceCell)
          masterPin = getAssociatedPin(point de contact);
          drive = drive + masterPin.getNet().getInputDrive();

        ELSE
          alpha = 0;

          IF((point de contact).getEquation().isFromExistingLibrary())
            alpha = findAlphaFromAttributesList();

            IF(alpha == 0)
              alpha = getAlphaFromModel();
            ELSE
              alpha = findAlphaFromAssociatedModel();

          drive = drive + alpha;
      ELSE
        drive = drive + (point de contact).getResistance();
    ELSE
      CONTINUE;

  RETURN drive;
```

8

Figure 4.4 Algorithme pour la détermination de la résistance d'entrée

4.1.3 La propriété d'unacité

Une fonction peut être soit unate positive, soit unate négative, soit aucun des deux, qu'on nomme binate. Alors, Une sortie est dite unate positive par rapport à une entrée si le passage de 0 à 1 de l'entrée entraîne soit une transition de 0 à 1 de la sortie, soit une valeur constante de la sortie. Le **ET** logique et le **OU** logique sont des exemples de portes unates positives. Par contre, pour le même mouvement d'entrée, si la sortie passe de 1 à 0 ou que l'état reste constant, elle est unate négative. Le **NON** logique est un exemple de portes unates négatives. Lorsqu'on ne peut prédire le mouvement de la sortie, elle sera considérée comme binate. Le **XOR** est un exemple de portes binate. Pour une meilleure compréhension, ces relations sont à nouveau représentées dans le tableau 4.1.

Tableau 4.1 Propriétés d'unacité

Propriété	Changement à l'entrée	Changement à la sortie	Exemple
Unate +	$0 \longrightarrow 1$	$0 \longrightarrow 1$, constant	**ET, OU**
Unate -	$0 \longrightarrow 1$	$1 \longrightarrow 0$, constant	**NON**
Binate	$0 \longrightarrow 1$	$0 \longleftrightarrow 1$, constant	**XOR**

La connaissance de cette propriété est utile lors de la propagation des temps d'arrivée et des temps requis. En effet, pour chaque porte, les temps sur le front montant et sur le front descendant sont connus et il est ainsi facile de déterminer sur quel front à la sortie ces temps sont propagés. Pour les besoins du projet, cette propriété n'est utile que dans ce contexte, mais elle pourrait être utilisée pour d'autres modules subséquents. Pour plus d'informations sur cette propriété, les livres de [De Micheli, 1994],[Devadas *et al.*, 1994] et [Hurst *et al.*, 1985] sont un bon point de départ.

4.2 Propagation des temps et recherche des chemins critiques

Les temps d'arrivée indiquent le moment à partir duquel les entrées sont prêtes et utilisables. Cependant, les signaux ne sont pas émis tous en même temps. Selon l'environnement extérieur à un circuit, les temps d'arrivée peuvent différer beaucoup entre les différentes entrées. Les temps requis, eux, sont les temps qui sont demandés à la sortie pour que le circuit arrive à respecter les délais imposés. Si le flux temporel dépasse cette valeur, il faut revenir en arrière et repenser notre distribution des délais de sorte qu'on atteigne cette valeur.

Quant aux blocs internes, ceux-ci sont constitués des parties à l'intérieur du circuit qui sont reliées entre elles par des interconnexions. Chacun de ces blocs a un délai qui lui est propre et celui-ci vient influencer le délai total du flux temporel de l'entrée jusqu'à la sortie du circuit. Étant donné que la majorité des circuits numériques sont synchrones, les contraintes de délais dans les circuits sont habituellement liées aux périodes des différents signaux d'horloge utilisés. De plus, les chemins entre les éléments séquentiels peuvent contraindre l'horloge du système global et, ainsi, affecter la performance de l'ensemble du circuit. Ici, on entend par chemin critique un chemin dont la flexibilité totale sur le chemin est négative.

4.2.1 Les temps d'arrivée

En premier lieu, pour faire la propagation des temps d'arrivée, ceux-ci doivent être assignés aux entrées principales du circuit évalué. Pour ce faire, lors de la lecture de la grammaire, des instructions sont prévues à cet effet. À l'intérieur de la fonction même, une fois les équations et les instances établies, les mots-clés «SET ARRIVAL TIME» et «SET REQUIRED TIME» suivis de l'indicateur que le signal est en montée ou en descente, «RISE» ou «FALL», et le nom du point de contact seront utilisés pour assigner une valeur numérique à ce point de contact.

La figure 4.5 représente l'action de propager des temps d'arrivée dans un circuit. Principalement, pour chaque entrée du circuit, les temps d'arrivée sont «poussés» dans le circuit jusqu'à ce que l'on arrive aux sorties ou à une entrée d'une porte séquentielle. Dans ce cas, une nouvelle propagation sera effectuée des sorties de la porte séquentielle avec un temps d'arrivée de 0. Pour l'instant, le système ne tient pas compte des temps d'établissement, de maintien et de basculement, mais la structure d'annotations permet de rajouter ces éléments. L'horloge associée à une porte séquentielle déterminera le temps d'arrivée à cette porte.

En allant plus en détails dans le processus, pour chaque point de contact d'entrée, l'interconnexion sur laquelle il se trouve peut être déterminée et pour tous les points de contact sur cette interconnexion, le temps d'arrivéé lui est assigné. Il est évident que, plus l'interconnexion est longue et plus le délai de transmission est élevé, mais, pour la version initiale du système, les annotations sur les interconnexions ont été mises de côté. Donc, lorsque qu'on assigne le temps d'arrivée au point de contact, il faut spécifier ici qu'il est assigné à l'annotation de ce point de contact se trouvant dans la table d'annotation. Ensuite, selon que la cellule suivante soit une instance ou une équation, les opérations effectuées seront différentes.

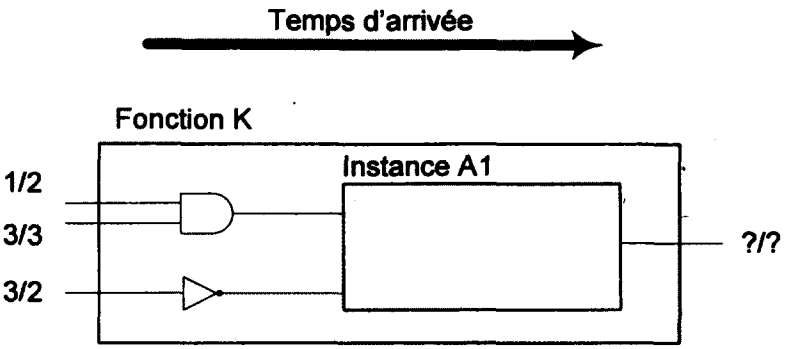


Figure 4.5 Propagation des temps d'arrivée

Propagation dans une instance

Si la cellule rencontrée est une instance, il faut d'abord remplir la table qui contient les objets décrivant les chemins possibles au travers l'instance. Cette table sera utile lors du calcul du chemin critique. Ensuite, en extrayant la fonction instanciée, le même processus de propagation des temps d'arrivée est appliqué. Toutefois, avant de pouvoir faire une propagation des temps d'arrivée, les contraintes sur l'instance doivent être établies et sauvegardées dans la table d'annotation. Lors de cette opération, une table appartenant à la fonction de référence et indiquant tous les chemins possibles à partir d'une entrée donnée sera remplie. Ceci est pour éviter de recalculer, pour chaque instance d'une même fonction, si une entrée est liée à une sortie. Une fois terminé, on revient au niveau supérieur et, pour chaque point de contact en sortie sur l'instance, la propagation se fait sur l'interconnexion associée au point de contact.

Propagation dans une équation

Par contre, si la cellule rencontrée est une équation, le processus est un peu différent. Une table de délai, identique à celle utilisée pour l'instance à la différence que celle-ci pointe vers une valeur numérique du délai entre une entrée et une sortie, est associée à l'équation dans la table d'annotation. Celle-ci sera remplie au fur et à mesure que les valeurs de délais seront calculées pour l'entrée et la sortie donnée. Pour calculer le délai, deux possibilités s'offrent à nous : l'équation peut provenir d'une bibliothèque existante ou non.

Lorsqu'elle provient d'une bibliothèque existante, le calcul du délai se fait avec le modèle qui est associé à la porte logique. Une fois le modèle déterminé, il ne reste plus qu'à appliquer l'équation du modèle. Lorsque la valeur est représenté par le terme «SUM», cela indique que la valeur est égale à la somme des capacités sur l'interconnexion. Advenant le

fait que l'équation du délai soit du type «linéaire par morceau», le calcul ne s'effectue pas de la même manière qu'une équation qui reste la même. Comme mentionné précédemment, le modèle linéaire par morceau est conçu pour supporter plusieurs paramètres, mais le traitement du délai ne se fera que s'il n'y a seulement que des coordonnées simple, donc dans le plan XY.

Alors, si la valeur est une valeur définie par l'équation, le délai qui lui est associé est automatiquement choisi. Par contre, si elle est différente, nous devons alors faire une approximation par interpolation du délai avec l'équation 4.1 présentée ci-dessous. En déterminant la pente entre deux coordonnées (membre gauche de l'équation), on trouve la valeur de délai donnée par y_δ en l'isolant d'un des deux membres de droite. Cette valeur de délai est associé à la valeur de la charge à la sortie, représentée par x_δ.

$$\frac{y_{\delta+i} - y_{\delta-i}}{x_{\delta+i} - x_{\delta-i}} = \frac{y_{\delta+i} - y_\delta}{x_{\delta+i} - x_\delta} = \frac{y_\delta - y_{\delta-i}}{x_\delta - x_{\delta-i}} \tag{4.1}$$

S'il s'agit d'une équation qui ne fait pas partie d'une bibliothèque existante, celle-ci ne possède donc pas de modèle propre. Pour lui définir un modèle, plusieurs possibilités existaient, mais l'attribution d'un modèle selon le nombre d'entrée de la fonction a été choisi. Alors, une bibliothèque spécifique à ce type d'équation a été construite de façon à pouvoir assigner des modèles de délais, ainsi que des valeurs de capacités pour les différents points de contact, en fonction du nombre d'entrée. Il s'agit bien ici d'approximer les valeurs de délais provenant de l'équation puisqu'il n'est pas possible, à ce stade, de les connaître.

Une fois le délai calculé, il ne reste plus qu'à continuer la propagation des temps d'arrivée. C'est ici que la propriété d'unacité devient utile. Mais, tout d'abord, une vérification est effectuée pour s'assurer que toutes les entrées ont une valeur de temps. Ensuite, trois possibilités sont présentes puisque l'équation peut être positivement unate, négativement unate ou binate.

1. Si elle est positivement unate, le temps à la sortie, peu importe le sens, sera égal à la pire somme calculée entre le temps d'arrivée à l'entrée de l'équation et son délai associé.

2. Si elle est négativement unate, la valeur à la sortie est la même, mais elle est sauvegardée sur le sens inverse de celui de l'entrée.

3. Si elle est binate, le calcul est légèrement plus complexe. La valeur de temps à la sortie sera égale à la somme entre le délai maximum et la valeur maximale en entrée, représentée par l'équation 4.2.

$$t_{sortie} = t_{entrée_{MAX}} + \Delta_{MAX} \tag{4.2}$$

De plus, si l'équation est une équation pour une porte séquentielle, un traitement particulier lui est réservé. En effet, le temps d'arrivée à la sortie ne sera pas reflété par le modèle de délai, mais bien par l'horloge utilisée. Alors, la valeur attribuée à la sortie sera 0 puisque la valeur peut changer à chaque coup d'horloge. La valeur 0 implique qu'il n'y a qu'un seul signal d'horloge dans le circuit, et que les temps requis et les temps d'arrivée sont établis par rapport au front montant de l'horloge.

4.2.2 Les temps requis

En ce qui concerne la propagation des temps requis, le principe est le même, mais dans le sens inverse, c'est-à-dire des sorties vers les entrées. Ce processus peut être mieux illustré à l'aide de la figure 4.6. Également, pour les portes séquentielles, les temps requis aux entrées seront égaux à la période de l'horloge attachée à la porte.

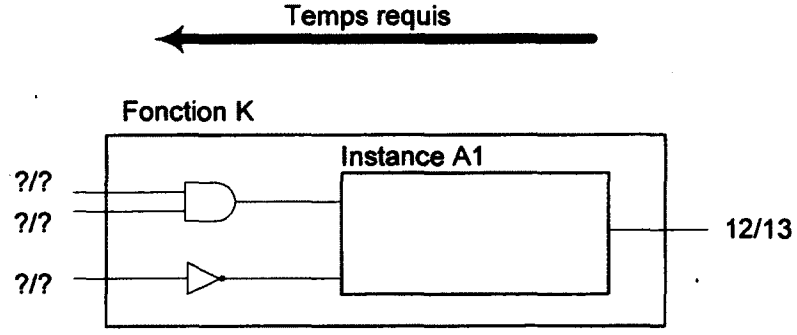


Figure 4.6 Propagation des temps requis

Comme pour les temps d'arrivée, pour chaque interconnexion reliée à un point de contact d'une sortie, on propage les temps requis qui ont été préalablement établis. Ici, contrairement à la propagation des temps d'arrivée, nous avons une interconnexion avec plusieurs points de contact propageant leurs temps requis vers un seul point de contact. Alors, il faut d'abord vérifier que tous les points de contact ont une valeur assignée dans la table d'annotation pour ensuite déterminer, en montée et en descente, la valeur de temps la plus critique. Celle-ci est la valeur la plus petite rencontrée.

La propagation des temps requis est une étape qui est beaucoup moins longue à exécuter, en partie grâce au fait qu'elle réutilise des parties déjà existantes comme les contraintes pour chaque instance et la table de délais pour chaque équation puisque ces éléments demeurent

inchangés. Par contre, en ce qui a trait aux équations et à l'unacité, si l'équation est binate, alors la valeur de temps à l'entrée sera égale à la somme de la valeur minimale obtenue en sortie avec le délai maximal entre les deux points de contact, représenté par l'équation 4.3

$$t_{entrée} = t_{sortie_{MIN}} + \Delta_{MAX} \tag{4.3}$$

De plus, encore une fois, lorsqu'une porte séqentielle est détectée, un traitement particulier est appliqué. En effet, la propagation des temps requis se termine sur la sortie d'une équation séquentielle et elle repart de chaque entrée avec une valeur égale à la période de l'horloge liée à cette équation.

Une fois la propagation des temps d'arrivée et des temps requis complétée, la flexibilité pour chaque point de contact du circuit en appliquant l'équation 4.4 est facilement obtenue. Tous les éléments nécessaires pour trouver les chemins critiques et faire de la gestion de budgets de délai sont présents.

$$Flexibilité = temps_{requis} - temps_{arrivée} \tag{4.4}$$

4.2.3 Recherche des chemins critiques

On appelle chemin critique le chemin ayant la flexibilité la plus petite dans le circuit. L'identification des chemins critiques permet de modifier certains éléments sur ce chemin afin d'améliorer sa flexibilité. Par exemple, à la figure 4.7, un chemin critique appartenant à un circuit est illustré. Avant de pouvoir trouver ce chemin critique, il faut d'abord connaître tous les points de contact pouvant être utilisés comme point d'entrée du chemin. Cela signifie les points de contact globaux du circuit ainsi que ceux provenant de la sortie d'une porte séquentielle. Alors, le circuit sera traversé afin de mettre dans une liste tous ces points de contact jumelés avec leur annotation propre. Pour les chemins provenant de portes séquentielles, le parcours utilisé, quoique semblable, est différent. Alors, une variante de la méthode principale sera utilisée traitant ce cas particulier.

Pour déterminer le chemin le plus critique, cette liste sera parcourue et le ou les points de contacts ayant la pire flexibilité seront conservés pour la suite des événements. Uune valeur de correction d'erreur ϵ est introduite afin de s'assurer que deux valeurs de flexibilité pouvant possiblement être la même soient toutes les deux considérées. La raison est que des erreurs d'arrondis peuvent se glisser dans le calcul des délais.

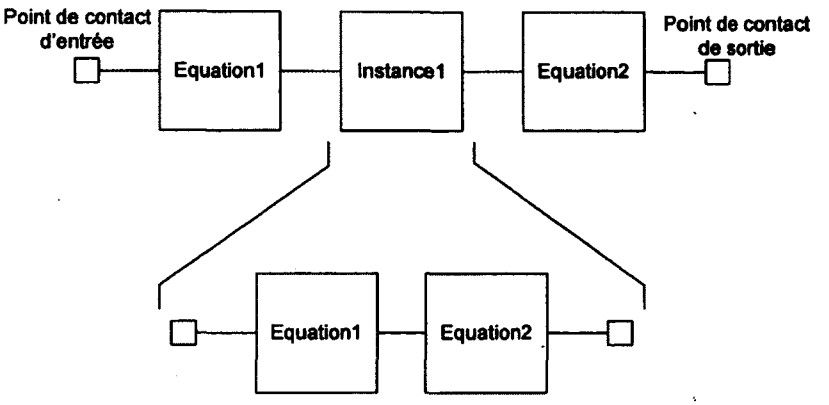


Figure 4.7 Chemin critique dans un circuit

L'objet utilisé pour représenter le chemin critique est composé de 5 éléments :

1. Un point de contact indiquant l'entrée ;
2. Un point de contact indiquant la sortie ;
3. L'indication du sens du signal (montée ou descente) entre les 2 points de contact ;
4. La flexibilité du chemin ;
5. Une liste d'objets (instances et équations) se trouvant entre les 2 points de contact.

Pour bien démontrer la hiérarchie, chaque objet de cette liste contient un champ qui, lui aussi, comprend tous les objets de l'entrée à la sortie de l'instance.

Alors, pour chaque point de contact dans la liste vue précédemment, un objet pour le chemin critique est créé et se voit placé dans une liste associée à l'instance de test du circuit. Les chemins critiques sont créés de façon itérative, en parcourant le circuit. Lors du traitement d'un nouveau segment, il existe deux possibilités :

1. Le segment est totalement nouveau et doit être créé et ajouté ;
2. Le point d'entrée du segment fait déjà partie d'un segment existant.

Si le premier cas est vrai, l'interconnexion liée au point de contact d'entrée est choisie et le prochain élément sur le chemin critique est utilisé. Sinon, c'est un chemin qui a été découvert en traversant un chemin critique, un chemin qui possède le même point d'entrée qu'un chemin déjà existant, mais qui peut posséder un point de sortie différent de celui-ci. Dans ce cas, on détermine le dernier objet sur ce chemin et on utilise le point de contact sur la sortie de cet objet pour exécuter la même opération.

Par la suite, les points de contact sur l'interconnexion ayant la même flexibilité que le point de contact d'origine sont déterminés. Si plus d'un point de contact est trouvé, un nouvel objet pour le chemin critique comprenant les mêmes éléments que l'original est créé puisque les deux chemins ont eu le même parcours jusqu'à ce point. Pour chaque point de contact ainsi trouvé, nous plongeons dans la cellule, puisqu'il s'agit d'une équation ou d'une instance, pour pouvoir ajouter celle-ci au chemin critique.

Lorsqu'il s'agit d'une équation, nous devons vérifier si celle-ci correspond à une porte séquentielle. Si c'est le cas, le traitement est très simple, car on ne fait que spécifier le point de contact en sortie du chemin avec le point de contact sur l'entrée de l'équation. Sinon, nous devons choisir dans la table de délai associée l'objet représentant le chemin emprunté et placer cet objet dans la liste des éléments se trouvant sur le chemin dans l'objet représentant le chemin critique.

S'il s'agit d'une instance, le principe est également le même, mais comporte quelques particularités. La première est qu'il faut entrer à l'intérieur de la fonction référencée et cette action doit retourner une liste d'éléments rencontrés entre l'entrée et la sortie de l'instance. Cette liste sera assignée à l'objet représentant ce chemin qui se trouve dans une table faisant partie de l'instance. De cette façon, la hiérarchie à même le chemin critique est conservée. Encore une fois, il est possible que d'autres chemins critiques aient été trouvés lors de l'entrée dans la fonction référencée. Ceux-ci se verront ajoutés à la liste de chemins critiques pour être ensuite traités.

Finalement, en sortant d'une instance ou d'une équation, une interconnexion est à nouveau rencontrée et le même processus continue de s'appliquer jusqu'à ce qu'un point de contact en sortie du circuit soit trouvé. Ayant obtenu les chemins critiques dans le circuit, le système est maintenant en mesure de faire de la gestion de budgets de délai.

Particularités à mentionner

De façon à obtenir un module de gestion de budgets de délai robuste, une implémentation en fonction des trois points suivants a été effectuée.

- Boucles : Les boucles se retrouvent principalement en présence d'objets séquentiels ou dans un système fermé. Pour éviter tout problème, la vérification des entrées se fait au niveau des équations, non au niveau des instances. Par là, on entend que la vérification à chaque niveau de hiérarchie n'est pas possible lorsque des boucles sont impliquées. Si une des entrées d'une instance est également une de ses sorties, les valeurs de temps ne peuvent pas être déterminées. En utilisant les équations, toutes les valeurs de temps, en entrée comme en sortie, sont nécessaires de façon à

continuer la propagation. C'est pourquoi une propagation des temps se fait au niveau des portes séquentielles.

- Objets binates : Les objets binates sont, bien sûr, les instances, mais aussi à l'occasion des équations logiques. Étant donné la particularité du calcul de délai pour ce type d'objet, chaque signal d'entrée offre deux possibilités à la sortie. Donc, un traitement adéquat de ces objets est effectué lors du calcul des chemins critiques.

- Entrées constantes : Il peut arriver parfois que les entrées d'instance ou d'équation peuvent être constantes, c'est-à-dire qu'elles ne sont pas attachées à la fonction. On se réfère ici à la mise à la terre et à l'alimentation. Étant donné que ces entrées n'impliquent aucun délai, un traitment spécial a lieu dès leur détection. La hiérarchie dans le système, ici aussi, complique légèrement le problème puisqu'il faut s'assurer que le signal se rende à la porte logique afin d'effectuer les calculs de délais dans celle-ci.

4.3 Stratégies de gestion des délais employées

À l'aide des chemins critiques déterminés selon la méthode décrite plus haut, il est possible de déterminer les budgets de délai qui pourront être employés pour la synthèse. De nombreuses stratégies de gestion de délai sont possibles. Dans ce qui suit, deux méthodes sont décrites : la gestion utilisant des budgets de délai uniformes et la gestion utilisant des budgets de délai fractionnels.

4.3.1 Gestion de budgets uniformes

La figure 4.8 représente l'algorithme d'une gestion de budgets de délai faite de façon uniforme. La vérification faite au début permet de déterminer si un budget de délai a déjà été assigné à l'élément du chemin critique. Si c'est le cas, on soustrait ce budget de la valeur de la flexibilité totale du chemin critique. Dans le cas contraire, l'élément est ajouté à une liste de traitement. Une fois la vérification terminée, la longueur de cette liste déterminera la valeur de budget à assigner à chaque élément. Cet algorithme de gestion fait également usage de la hiérarchie dans le circuit. Donc, pour un budget quelconque assigné à une instance de fonction, ce budget sera divisé à part égale entre les éléments du chemin critique dans cette instance comme le montre la figure 4.9.

```
applyUniformSlack()

  uniformSlack = slack;

  FOR(chaque élément sur le chemin critique)

    IF(cellule est de type InstanceCell)
      IF(élément possède un budget)
        uniformSlack = uniformSlack - budget de instance;
      ELSE
        ajouter à liste de traitement;
    ELSE
      IF(élément possède un budget)
        uniformSlack = uniformSlack - budget de équation;
      ELSE
        ajouter à liste de traitement;

    uniformSlack = uniformSlack / taille de la liste de traitement;

    FOR(chaque élément de la liste de traitement)

      IF(cellule est de type InstanceCell)

        associer instance avec son slack;
        applyUniformSlack();

      ELSE
        associer équation avec son slack;
```

Figure 4.8 Algorithme de gestion de budgets uniformes

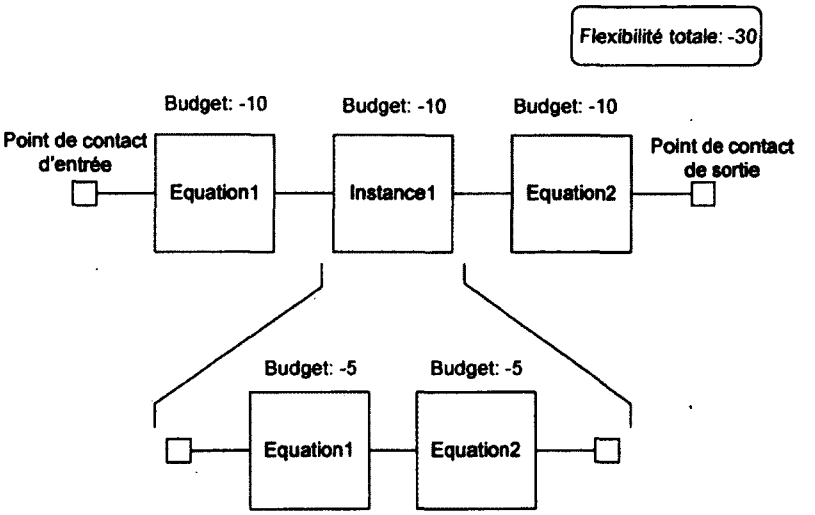


Figure 4.9 Hiérarchie et budgets de délai

4.3.2 Gestion de budgets fractionels

La figure 4.10 représente l'algorithme d'une gestion de budgets de délai faite en relation avec le délai de chaque porte logique sur le chemin critique. Le délai total sur le chemin critique est d'abord trouvé. Ensuite, pour respecter la hiérarchie, si l'élément du chemin critique est une instance, il faut exécuter cette opération à l'intérieur même de l'instance pour obtenir le budget de délai assigné à cette instance. Lorsqu'il s'agit d'une équation, le budget assigné sera égal à une fraction de la flexibilité totale, cette fraction étant déterminée par le rapport *délai spécifique/délai total.*

```
applyRatioSlack()

  totalDelay = getTotalDelay();

  FOR(chaque élément sur le chemin critique)
    IF(cellule est de type InstanceCell)

      returnRatio = applyRatioSlack();
      associer returnRatio à instance;

    ELSE

      specificDelay = équation.getDelay();
      ratioSlack = slack * specificDelay / totalDelay;
      associer ratioSlack à équation;
```

Figure 4.10 Algorithme de gestion de budgets fractionnels

La flexibilité pour tous les éléments du système a pu être déterminée en effectuant le calcul des temps d'arrivées et des temps requis, et ce, de façon hiérarchique. De cette façon, les chemins critiques peuvent être déterminés et des stratégies de gestion de budgets de délai peuvent y être appliqués. Ceci complète le système, qui offre ainsi tout le nécessaire pour faire de la synthèse de circuits numériques. Il ne reste plus qu'à valider le système à l'aide d'un circuit de grande taille et à évaluer la qualité du système logiciel, ce qui est présenté dans le prochain chapitre.

CHAPITRE 5

TESTS ET ANALYSE

L'ensemble des tests qui seront effectués pour démontrer les capacités de notre plate-forme de synthèse est maintenant défini.

5.1 Explication des paramètres de tests

Afin de tester les deux stratégies de gestion de budgets de délai présentées dans la section précédente, un circuit de test contenant quelques milliers de portes logiques a été utilisé. Ce circuit est une représentation partielle d'un microprocesseur de 32 bits et est illustré à la figure 5.1. Il est composé de quatre sections distinctes : les registres de données, l'unité arithmétique et logique (ALU), la mémoire et les circuits de réécriture des données (*«write-back»*). Ces sections forment ce qui est appelé le chemin de données (*«datapath»*). Étant donné que ce circuit est quasi-fermé, des sondes ont été insérées pour chaque bit sortant de l'ALU et de la mémoire, ce qui nous donne 65 sorties. Pour obtenir un microprocesseur complet, quelques sections doivent être ajoutées comme les registres d'instructions, les registres d'adresses, les registres de décodage et le contrôleur, qui est généralement une machine à états finis. De plus, l'ALU décrite ici ne contient pas de logique pour faire la multiplication et la division.

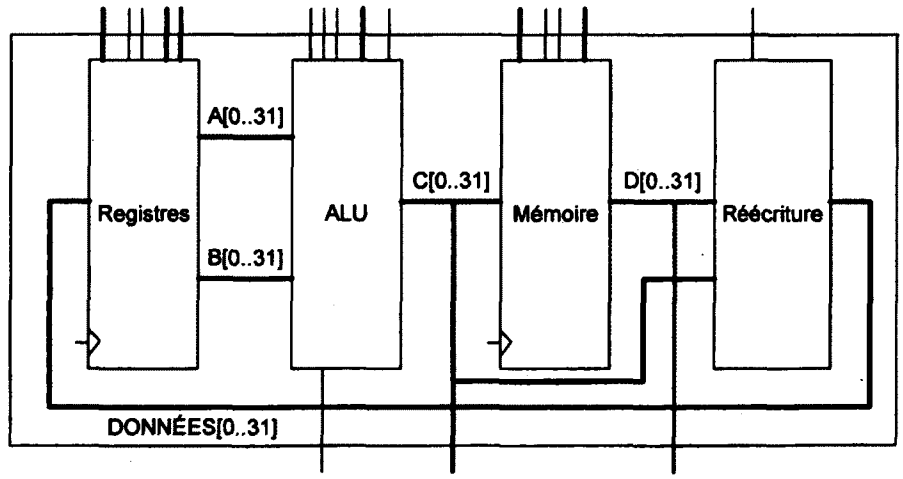


Figure 5.1 Le circuit de test

Pour expliquer chaque section du microprocesseur, une tranche de 1 bit est utilisée afin d'alléger les figures puisque chacune des tranches est assez similaire. La figure 5.2 montre la constitution des registres de données. Trois bits qui entrent dans le décodeur vont permettre à celui-ci de choisir lequel des 8 registres est utilisé pour le stockage de la donnée. Chacun de ces registres est attaché à l'horloge puisque ce sont des portes séquentielles. Un bit de remise à zéro et un bit de permission sont également utilisés. Le bit de permission (en anglais, «*enable*») donne ou refuse l'accès aux registres. La sortie des registres se dirige ensuite vers deux multiplexeurs à 8 entrées. Chacun d'eux a trois bits de sélection et leur sortie est envoyée à l'ALU pour le traitement arithmétique et logique.

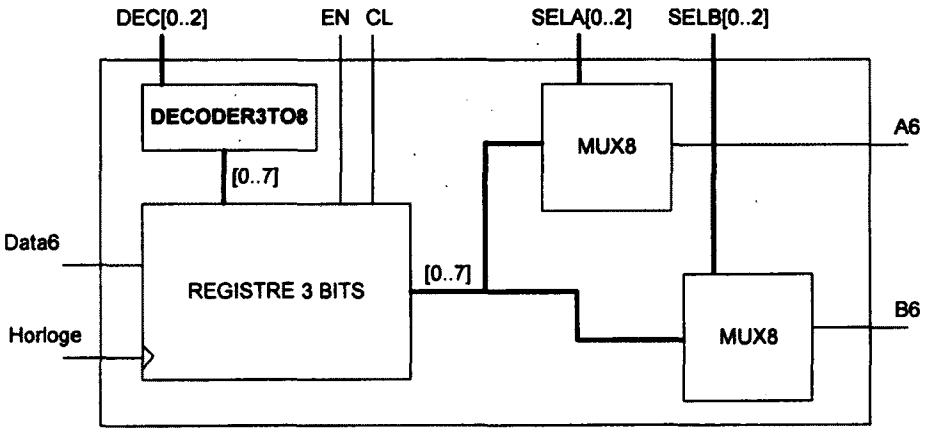


Figure 5.2 Les registres

L'ALU représentée par la figure 5.3 est composée de plusieurs blocs distincts. Les deux entrées sont premièrement inversées et le choix entre celles-ci et les entrées originales est déterminé à l'aide d'un multiplexeur. Ensuite, un traitement logique sur les deux entrées à l'aide d'une porte **ET**, d'une porte **OU** et d'une porte **XOR**. Le traitement arithmétique va s'effectuer sous la forme d'une somme des entrées avec la retenue («*CarryIn*») provenant de la tranche précédente. Cet additionneur est un additionneur parallèle à propagation de retenue («*ripple carry*»). La raison pour laquelle celui-ci a été choisi est pour permettre au chemin critique de traverser plusieurs tranches de 1 bit du chemin de donnée, et ainsi de bien valider le calcul des délais, des chemins critiques et des budgets de délai.

Les sorties sont la retenue qui est utilisée par la tranche suivante et le résultat de l'addition. Tous ces résultats, en plus des entrées originales de l'ALU et deux bits de décalage, vont dans un multiplexeur à 8 entrées. Ces deux bits de décalage représentent le bit de gauche et celui de droite de la première entrée. Or, s'il n'y pas de bit de décalage présent comme dans le cas du bit 0 ou du bit 31, la valeur 0 est envoyée. Alors, la sortie de ce multiplexeur

subit, elle aussi, le même traitement que les entrées, c'est-à-dire qu'un choix entre sa vraie valeur ou sa valeur inversée sera fait à l'aide d'un autre multiplexeur.

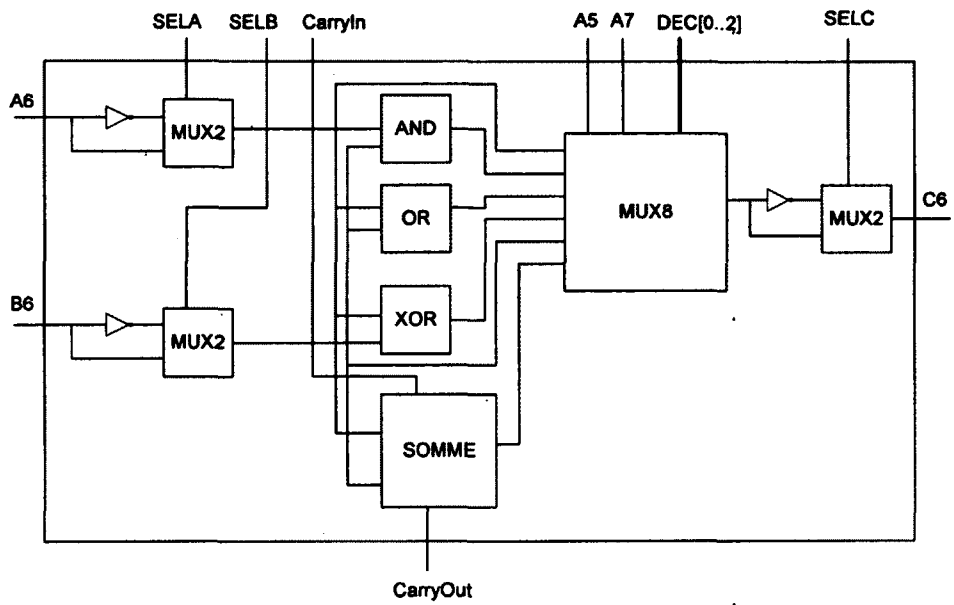


Figure 5.3 L'unité arithmétique et logique (ALU)

La mémoire utilisée comporte 128 mots de 32 bits, qui sont sélectionnés à l'aide d'une adresse de 7 bits, tel qu'illustré à la figure 5.4. Deux bits, les bits 6 et 7, qui entrent dans le décodeur vont déterminer dans lequel des quatre blocs de mémoire de 5 bits la donnée provenant de l'ALU va s'enregistrer. Un bit de permission est utilisé pour contrôler l'écriture dans la mémoire. Les bits 0 à 5, ceux servant au décodage et ceux servant à la lecture, sont attachés à la mémoire de 5 bits, en plus du bit de permission, du bit de mise à zéro, de l'horloge et de l'entrée provenant de l'ALU. Deux bits, les bits 6 et 7, servant à la lecture de la mémoire vont déterminer la sortie des quatre modules de mémoire de 5 bits devant être lue avec l'aide d'un multiplexeur. Le module de mémoire de 5 bits est semblable à celui de 7 bits, mais celui de 3 bits ressemble davantage à celui décrit pour les registres de donnée, avec un multiplexeur en moins puisque nous n'avons qu'une seule sortie.

Enfin, la section portant sur la réécriture des données est la plus légère en termes de portes logiques. Illustrée à la figure 5.5, elle est constituée d'un multiplexeur à 2 entrées, la sortie de la mémoire et la sortie de l'ALU, et la sortie se dirige vers les registres de donnée.

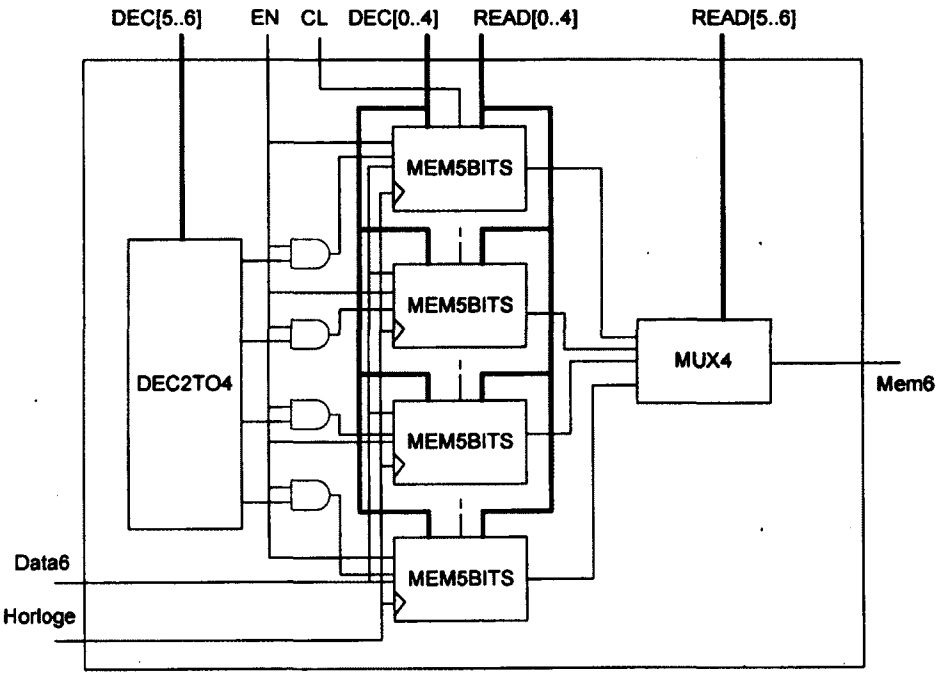


Figure 5.4 La mémoire

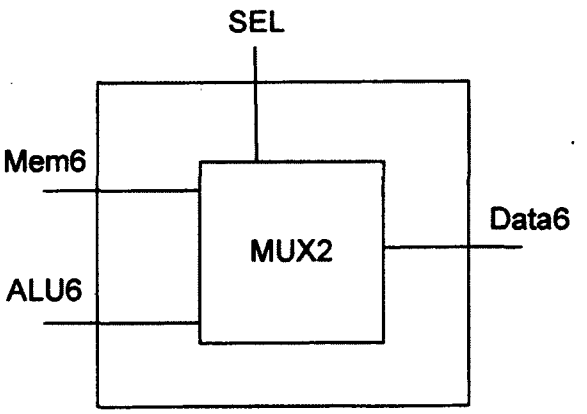


Figure 5.5 Le circuit de réécriture (*«write-back»*)

5.2 Analyse des tests effectués

Afin de faire de la gestion de budgets de délai, il faut déterminer les temps d'arrivée et les temps requis pour toutes les entrées et sorties du circuit. Afin de s'assurer une flexibilité négative, les entrées et les sorties ont toutes été initialisées avec la même valeur, soit 1 ns. Modifier les temps d'arrivée et les temps requis permet d'obtenir différents chemins critiques que ceux obtenus ici.

Le tableau 5.1 montre le nombre de points de contact, d'interconnexions distinctes, de portes logiques et de portes séquentielles pour chaque grande section du circuit de test et le nombre total de ces éléments pour le circuit en entier. On entend par interconnexion distincte une interconnexion ignorant la hiérarchie. Les points de contacts des connecteurs attachés aux instances ne sont pas compris dans les résultats présentés puisqu'ils ne sont là qu'à titre représentatif. Trois points peuvent être soulevés quant à la nature de ces résultats.

1. À elle seule, la mémoire représente près de 90% du circuit. Normalement, la mémoire ne fait pas partie des tests d'optimisation de délai. Cependant, de façon à tester rigoureusement les limites du système, elle a été incluse.

2. Ce tableau comporte deux valeurs particulières : les valeurs pour les points de contact et les interconnexions du circuit global. En effet, il y a 36 entrées principales, donc 36 points de contact et interconnexions supplémentaires. Il y a également 65 sorties principales, mais cela ne fait qu'ajouter au nombre de points de contacts. De plus, la mise à la terre (le 0 logique) a été utilisé à quelques endroits, ce qui explique l'interconnexion additionnelle.

3. Le *fanout* maximal sur une interconnexion est de 4352, soit le nombre total d'éléments séquentiels dans le circuit. La raison est que l'horloge est reliée à chacune de ces portes séquentielles et qu'elle est unique pour tout le circuit. Évidemment, l'ajout d'un arbre d'horloge («*clock tree*») diminuerait grandement cette valeur, mais le circuit de test a été conçu dans le but de valider le fonctionnement en présence de très grand fanout.

La figure 5.6 indique où sont situés les chemins critiques, obtenus lors des tests, dans le chemin de données. Après exécution, le système indique la présence de 32 chemins critiques, dont la flexibilité est de -5698. Ces 32 chemins ont tous la même origine, le bit 0 du signal de lecture de la mémoire. L'état du signal d'entrée est également le même pour tous les chemins, soit en montée. Par contre, l'élément qui détermine le nombre de

Tableau 5.1 Éléments composant le circuit de test

	Points de contact	Interconnexions	P. log.	P. séq.
Registres	9920	2560	2304	256
ALU	5568	1568	1568	—
Mémoire	135308	35842	31746	4096
Réécriture	576	160	160	—
Circuit principal	101	37	—	—
Total	**151473**	**40167**	35778	4352

chemins critiques a été la charge appliquée à la sortie. Plus la charge est élevée et plus le délai engendré est élevé. Ainsi, nous obtenons 8 interconnexions ayant la plus grande charge appliquée.

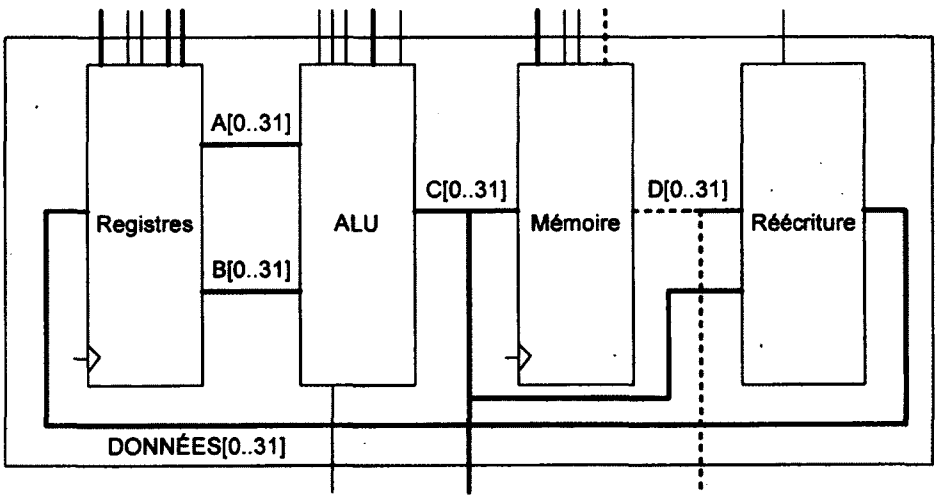


Figure 5.6 Les chemins critiques dans le circuit

La figure 5.7 permet de mieux comprendre la présence de 32 chemins critiques au lieu de 8. Le délai dans les portes AND4 est le même puisque le même point de contact d'entrée est utilisé pour les 4 portes. De plus, la porte OR8 ne provient pas d'une cellule existante : chaque entrée possède le même délai.

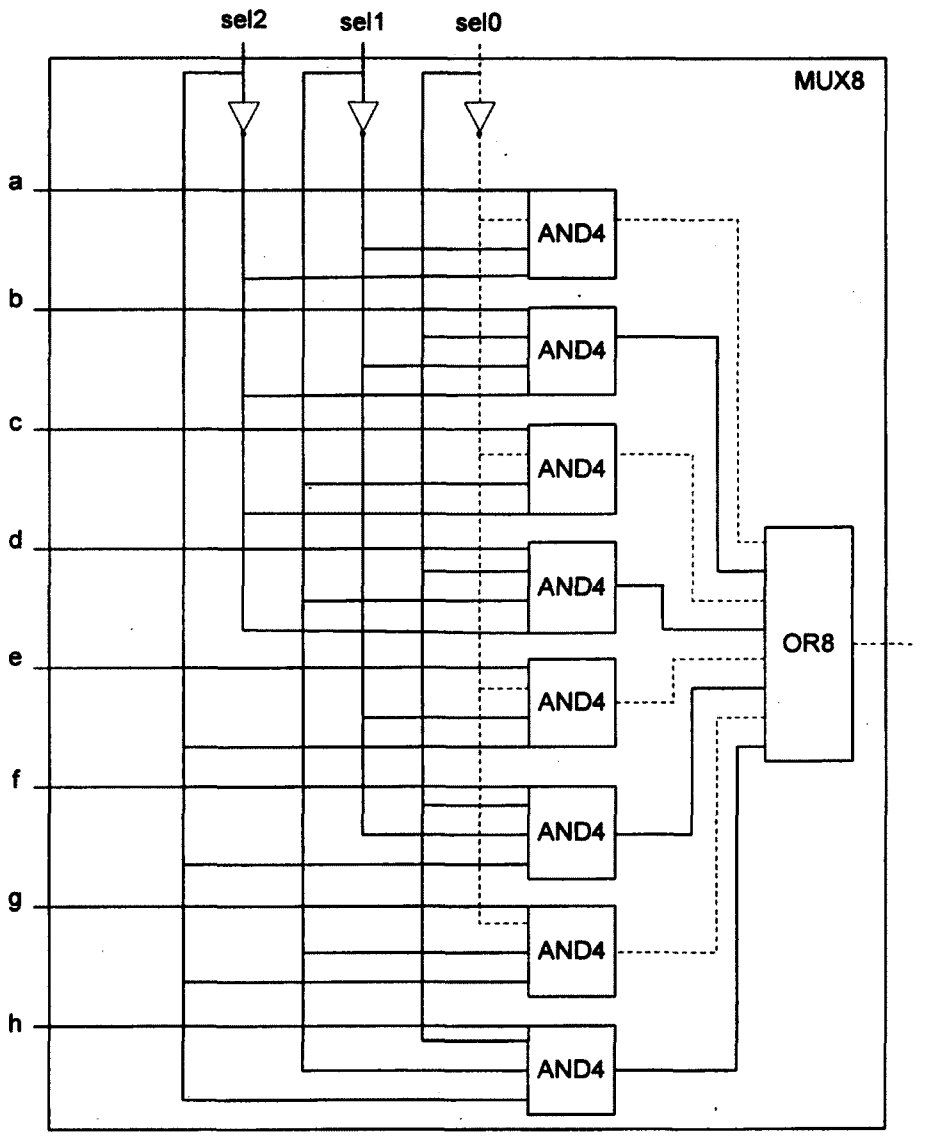


Figure 5.7 Une vue détaillée d'un des chemins critiques

La figure 5.8 représente un chemin critique potentiel à l'intérieur d'un circuit. Ce chemin comprend une instance de fonction, mais ce qui le caractérise davantage est qu'il commence à la sortie d'une porte séquentielle et se termine à l'entrée d'une autre porte séquentielle. Un chemin critique est toujours déterminé sur une logique combinatoire reliant des portes séquentielles ou des points de contacts globaux du circuit. Les chemins critiques obtenus ne contiennent pas de portes séquentielles comme le montre la figure 5.6.

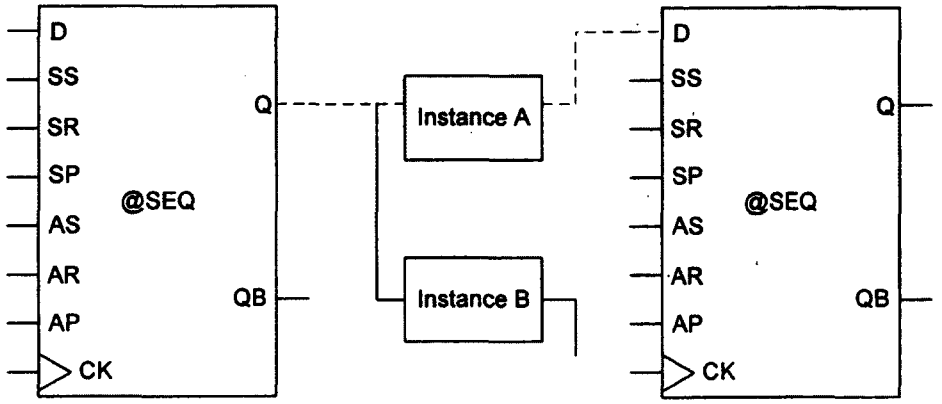


Figure 5.8 Chemin critique avec portes séquentielles

Le tableau 5.2 indique les caractéristiques des chemins critiques trouvés par le système. On remarque que ces chemins démarrent tous d'une entrée principale unique, mais qu'il existe 8 possibilités pour la sortie, toutes reliées à la mémoire. On remarque aussi que les chemins critiques traversent l'ensemble du circuit, parcourant 7 niveaux de hiérarchie. Dans ces 7 niveaux de hiérarchie, 13 instances et 7 équations ont été détectées. Enfin, la raison pour laquelle l'état du signal de sortie est différent de celui de l'entrée est qu'un inverseur a été rencontré sur le chemin. Comme il a été vu précédemment, les inverseurs sont unates négatifs ce qui change l'état du signal. Le changement de phase du signal le long du chemin critique valide ainsi le fonctionnement du système en présence de portes complexes.

L'analyse de la couverture permet de déterminer quelles sections de code ne sont pas utilisées et, ainsi, faire des tests pour que ces sections soient utilisées. Pour voir la couverture de chaque classe, se référer aux figures B.1, B.2, B.3 et B.4 à l'Annexe B. Le calcul de la couverture a été effectué avec le «*plugin*» eCobertura pour la plate-forme de développement Eclipse. On peut remarquer que la couverture totale est de 82%, ce qui est très acceptable. Plusieurs raisons expliquent cette couverture :

Tableau 5.2 Propriétés des chemins critiques

Nombre de chemins critiques	32
Interconnexion d'entrée	MEM_RE0
Interconnexions de sortie	mem2, mem6, mem10, mem14, mem18, mem22, mem26, mem30
État de l'entrée	Montée
États des sorties	Descente
Nombre d'instances	13
Nombre d'équations	7
Niveaux de hiérarchie	7
Flexibilité totale	-5698.1

- La plupart des classes implémente l'interface *Cloneable* pour le clonage, mais certaines d'entre elles ne se font jamais cloner dans l'exemple complexe utilisé pour valider la plate-forme de synthèse.

- Le code touchant aux entrées/sorties et aux éléments de logique à 3 états n'est pas couvert par le circuit de test puisque celui-ci n'en contient pas.

- La plate-forme de synthèse actuelle ne peut supporter pour l'instant que 4 types de modèle de délai. Cependant, les modèles quadratique et exponentiel n'ont pas été utilisés. Alors, les opérations couvrant spécifiquement ces modèles ne sont pas couvertes.

- Le traitement d'erreur est une autre cause de manque de couverture. Le circuit complexe utilisé pour valider le système (un sous-ensemble d'un processeur 32 bits) a été décrit sans erreur de syntaxe. Puisque le traitement des erreurs de syntaxe est généré automatiquement par ANTLR, sa validation n'était pas une priorité.

- Pour l'instant, l'assignation de modèles de délai à une équation logique n'appartenant pas à une cellule prédéfinie se fait de façon arbitraire. Seulement quelques modèles ont été assigné à des équations, ce qui amène un manque de couverture à cet endroit.

Le «*plugin*» Metrics a également été utilisé pour analyser la structure du code en se basant sur quelques métriques[1] et les résultats sont présentés à la figure 5.9. On peut remarquer que le nombre de classes dans le système est de 150 et le nombre de méthodes s'élève à 750, mais la plupart de ces classes et méthodes sont générées par ANTLR. Aussi, le système comporte plus de 16000 lignes de code dont près de 13000 font partie de méthodes. On peut également s'apercevoir que trois métriques ne rencontrent pas les critères définis.

[1]Pour plus d'informations sur les métriques, se référer à [Henderson-Sellers, 1996].

- Le nombre de paramètres maximal pour une méthode étant de 5, huit classes ne la respectent pas. Des méthodes liées au calcul des chemins critiques et à l'impression des données contiennent en majorité 6 paramètres, mais des méthodes liées au traitement de l'arbre d'équation du modèle de délai varient entre 8 et 10 paramètres.

- Le nombre cyclomatique de McCabe est une mesure de la complexité des méthodes et le critère fixé par le métrique est de 10. Plus le nombre de branchement dans la méthode est grand, plus ce nombre sera grand. Évidemment, toutes les classes générées par ANTLR sont en défaut puisque le décomposeur analytique comprend majoritairement des branchements. Pour les autres classes touchées, la refactorisation devient un moyen afin de rencontrer le critère établi.

- La profondeur des blocs dans une méthode ne doit pas dépasser 5 pour satisfaire au critère. Les classes générées par ANTLR sont, encore une fois, principalement ciblées par ce métrique. Par contre, d'autres classes sont également touchées et la profondeur varient entre 6 et 7. Ici aussi, la refactorisation peut permettre de rencontrer le critère.

En somme, il a été démontré que la plate-forme est capable de traiter un circuit de test complexe en redonnant les chemins critiques qui y sont présents. De plus, étant donné que la règle empirique qu'un logiciel couvert à plus de 75% est habituellement reconnu comme étant un logiciel robuste et acceptable, la couverture obtenue par notre plate-forme, 82%, est très bonne. On peut aussi affirmer que le code est assez bien structuré, rencontrant quelques écarts à quelques endroits, ceux-ci étant majoritairement concentrés dans les classes générées. Enfin, il faut s'assurer de bien avoir répondu au problème présenté en résumant les étapes de conception de la plate-forme de synthèse et de se questionner sur ce que cette plate-forme peut apporter pour les futurs travaux dans le domaine.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per	100	0.667	0.525	2	/Synthesis/src/logic/Eval_Library.java	
Number of Attributes (avg/max per type)	280	1.867	2.526	18	/Synthesis/src/logic/Eval_Library.java	
Number of Children (avg/max per type)	12	0.08	0.77	9	/Synthesis/src/logic/Tools.java	
Number of Classes (avg/max per packageFragi	150	150	0	150	/Synthesis/src/logic	
Method Lines of Code (avg/max per method)	13030	17.235	35.72	330	/Synthesis/src/logic/LibraryParser.java	stmt
Number of Methods (avg/max per type)	750	5	9.657	74	/Synthesis/src/logic/BooleanNetworkCell.java	
Nested Block Depth (avg/max per method)		2.176	1.665	8	/Synthesis/src/logic/LogicParser.java	library
Depth of Inheritance Tree (avg/max per type)		2.287	0.897	3	/Synthesis/src/logic/InstanceCell.java	
Number of Packages	1					
Afferent Coupling (avg/max per packageFragn		0	0	0	/Synthesis/src/logic	
Number of Interfaces (avg/max per packageFr	2	2	0	2	/Synthesis/src/logic	
McCabe Cyclomatic Complexity (avg/max per		3.521	5.526	66	/Synthesis/src/logic/LogicParser.java	stmt
Total Lines of Code	16421					
Instability (avg/max per packageFragment)		1	0	1	/Synthesis/src/logic	
Number of Parameters (avg/max per method)		0.95	1.548	10	/Synthesis/src/logic/Model.java	executeAdd
Lack of Cohesion of Methods (avg/max per typ		0.155	0.304	1	/Synthesis/src/logic/LogicLexer.java	
Efferent Coupling (avg/max per packageFragm		14	0	14	/Synthesis/src/logic	
Number of Static Methods (avg/max per type)	6	0.04	0.255	2	/Synthesis/src/logic/BooleanNetworkRepository.java	
Normalized Distance (avg/max per packageFra		0.02	0	0.02	/Synthesis/src/logic	
Abstractness (avg/max per packageFragment)		0.02	0	0.02	/Synthesis/src/logic	
Specialization Index (avg/max per type)		1.578	1.439	3	/Synthesis/src/logic/Eval_Library.java	
Weighted methods per Class (avg/max per typ	2662	17.747	52.073	390	/Synthesis/src/logic/LogicParser.java	
Number of Static Attributes (avg/max per type	849	5.66	29.737	226	/Synthesis/src/logic/LibraryParser.java	

Figure 5.9 Les métriques utilisées

CHAPITRE 6

CONCLUSION

En conclusion, un rappel des points importants de ce projet de recherche est effectué pour ensuite terminer avec quelques pistes de recherche pour l'avenir.

6.1 Sommaire

En tenant compte de la problématique, une plate-forme de synthèse de circuits programmée en Java, ce qui remplit les conditions de l'utilisation d'un langage orienté-objet pour sa conception et de la portabilité de cette plate-forme, a été présentée. On a démontré également que cette plate-forme permet l'intégration de modules subséquents avec l'ajout d'un module de gestion de budget de délai au système. De plus, cette plate-forme a été conçue dans le but premier de faire de la synthèse axée sur les délais dans les interconnexions puisque le délai global est, de nos jours, associé davantage au délai dans les interconnexions qu'au délai dans les portes logiques.

Pour y arriver, plusieurs algorithmes existants ont été analysés afin de déterminer les besoins du système. Selon ces besoins, les règles et les conditions d'un langage de description de circuits logiques hiérarchique qui serviront à la lecture de la description logique provenant de la synthèse comportementale ont été énoncées. Deux ensembles de régles ont été créés : l'un pour les bibliothèques de cellules prédéfinies et l'autre pour la logique. Ces régles ont été écrites avec l'aide du générateur de décomposeur analytique ANTLR.

La plate-forme de conception, en lisant la description logique, est capable de générer un réseau booléen pour chaque fonction qu'elle rencontre et complète le réseau au fur et à mesure de cette lecture. De cette façon, on obtient une représentation interne du circuit analysé permettant la hiérarchisation. En plus de la logique combinatoire, la logique séquentielle et la logique à 3 états sont supportées. De façon à supporter les modifications hiérarchiques du réseau booléen, deux fonctions provenant de la synthèse logique, l'aplanissement et le groupement, ont été implémentées.

De façon à faire de la propagation de temps d'arrivée et de temps requis à travers les divers éléments du réseau booléen, plusieurs modèles de délais différents sont supportés (linéaire et linéaire par morceaux). En complément, le module de gestion de budgets de

délai permettant le calcul et l'identification de l'ensemble des contraintes temporelles du circuit traité a pu être attaché à la plate-forme principale.

Les tests ont été effectués sur une portion d'un microprocesseur de 32 bits, comportant principalement des registres, une ALU et une mémoire, qui a été conçue pour tester le module de gestion de budgets de délai. Deux stratégies ont été implémentées : l'une portant sur une gestion de budgets uniforme et l'autre portant sur une gestion de budgets relative au délai des portes logiques par rapport au délai global sur le chemin critique. Les résultats obtenus démontrent que le module de gestion de budgets de délai est fonctionnel et que la plate-forme est en mesure de lire une description logique et d'en créer une représentation interne sous la forme d'un réseau booléen.

6.2 Travaux futurs

L'ajout du module de gestion de budget de délai à la plate-forme principale était une partie importante de l'infrastructure, mais il existe plusieurs options pour les prochains ajouts. L'ajout d'un simulateur logique, qui envoie des signaux pouvant varier dans les entrées pour observer les sorties, peut se faire assez facilement. Ce simulateur permettrait d'observer les changements d'état des signaux en fonction du temps. Il serait ainsi possible de valider logiquement la fonctionnalité d'un circuit synthétisé par le système.

Puisque la synthèse physique s'exécute après la synthèse de haut niveau, un module pour le placement et un autre pour le routage peuvent être conçus et ajoutés à la plate-forme principale. Ces modules seraient en mesure d'utiliser les tables d'annotation pour ajouter les informations nécessaires aux divers éléments du circuit. De cette façon, les délais dans les interconnexions seraient optimisés et l'étape de synthèse logique, qui suit la synthèse physique dans le nouveau paradigme, permettrait d'optimiser les délais dans les portes logiques.

De nombreux algorithmes de synthèse logique existent et pourraient facilement être mis en oeuvre dans le système. En effet, la représentation hiérarchique, ainsi que l'utilisation d'une infrastructure complète de traversée du réseau booléen permettent la mise en oeuvre simple d'algorithmes de modification de circuits logiques. Entre autres, la mise en place de la technologie (en anglais, *«technology mapping»*) est une opération de la synthèse logique pouvant utiliser les résulats obtenus par la gestion de budget de délai. Elle permet de modifier certaines portes de façon à ce qu'elles rencontrent les contraintes de délai et permet également d'assigner des portes provenant de cellules pré-existantes à de la logique combinatoire et, ainsi, assigner des modèles de délai plus proches de la réalité.

Dans les circuits numériques, après la synthèse logique, il y a fréquemment des interconnexions qui relient un très grand nombre de portes en sortie. Dans le but de diminuer les délais sur ces interconnexions, il est essentiel d'insérer un ensemble de tampons («*buffers*»). Il existe de nombreux algorithmes de «*buffering*», dont le travail de maîtrise d'Amir Rabbani [Rabbani, 2007], et le système proposé, avec ses fonctionnalités élaborées pour représenter les interconnexions et les niveaux de hiérarchie, permettra une intégration aisée de ces algorithmes.

L'utilisation du langage Java pour le développement du système ouvre aussi la voie à une options intéressante : il serait possible de permettre l'accès au système via le web, en utilisant le Google Web Toolkit (GWT). Ceci permettrait de recadrer le système sous forme d'application web, où le fureteur deviendrait la fenêtre d'interaction et le serveur supporterait les calculs et la persistance des données. Cette organisation, selon le modèle de conception MVC («*Model-View-Controller*») distribué, rendrait possible l'utilisation à distance du système de synthèse.

ANNEXE A
LA FORME DE BACKUS-NAUR ÉTENDUE

La description logique des circuits doit se conformer à plusieurs règles syntaxiques pour que l'on puisse être capable de produire une représentation sous la forme d'un réseau booléen. Puisque la description a son propre langage, l'ensembles des règles définissant ce langage est considéré comme un métalangage. Ce métalangage, utilisé par ANTLR, est la forme de Backus-Naur étendue (EBNF pour *Extended Backus-Naur Form*) [Jinks, 2004]. Il faut noter ici que la forme étendue n'est pas supérieure à la forme standard ; elle ne fait qu'améliorer grandement la lisibilité du langage. De plus, plusieurs variantes d'EBNF existent, mais celle qui sera utilisée ici est celle employée par ANTLR.

Donc, la règle définie par ANTLR la plus simple serait celle-ci :

```
règle: définition;
```

et permet de dire que le membre de gauche égale le membre de droite. Si l'élément définissant la règle est lui-même une règle, on applique le même processus jusqu'à ce que le membre de droite ne référence plus une règle. S'il existe plus d'une définition pour une certaine règle, elles seront séparées par le caractère |. La partie étendue du langage provient de certains caractères améliorant sa lecture, présentée dans le tableau A.1. Enfin, pour différencier entre l'analyseur syntaxique et l'analyseur lexical, ANTLR impose la convention que les règles appartenant à l'analyseur lexical se doivent de débuter par une lettre majuscule.

Tableau A.1 Caractères pour l'EBNF [Luber, 2008]

Caractère	Description
()	Permet de grouper plusieurs éléments afin d'être considéré comme un seul jeton
?	Après le jeton, celui-ci se produit 0 ou 1 fois
*	Après le jeton, celui-ci se produit 0 ou plusieurs fois
+	Après le jeton, celui-ci se produit 1 ou plusieurs fois
.	Tout caractère ou jeton peut se produire 1 fois
..	Entre deux caractères, il permet d'englober tous les caractères se trouvant entre les bornes inclusivement

ANNEXE B
COUVERTURE SUR LE CODE

Cette annexe comporte la couverture totale de chaque classe du système de synthèse, incluant la couverture reliée aux branchements dans les méthodes.

Name	Lines	Total	%	Branches	Total	%
All Packages (2011-04-06 16:48:53)	8856	10798	82.02 %	2263	3432	65.94 %
logic	8856	10798	82.02 %	2263	3432	65.94 %
ASTLibraryExpression	46	57	80.70 %	21	36	58.33 %
ASTLibraryExpression$ExpressionEdge	3	4	75.00 %	0	0	-
ASTLibraryExpression$ExpressionNode	12	17	70.59 %	0	2	0.00 %
ASTLogicExpression	126	134	94.03 %	66	82	80.49 %
ASTLogicExpression$ExpressionEdge	3	9	33.33 %	0	0	-
ASTLogicExpression$ExpressionNode	20	32	62.50 %	4	8	50.00 %
AnnotationTable	10	10	100.00 %	0	0	-
Attribute	14	19	73.68 %	0	0	-
BooleanNetworkCell	577	656	87.96 %	270	340	79.41 %
BooleanNetworkRepository	56	62	90.32 %	18	22	81.82 %
Cell	107	134	79.85 %	46	70	65.71 %
Connector	25	27	92.59 %	4	4	100.00 %
Constants	0	0	-	0	0	-
Constraints	21	45	46.67 %	5	18	27.78 %
CriticalPath	101	126	80.16 %	33	50	66.00 %
Equation	169	207	81.64 %	66	91	72.53 %
EquationAnnotation	10	10	100.00 %	0	0	-
EquationCell	185	196	94.39 %	103	124	83.06 %
Eval_Library	1406	1819	77.30 %	199	307	64.82 %
Eval_Library$DFA11	11	12	91.67 %	0	0	-
Eval_Library$DFA27	11	12	91.67 %	0	0	-
Eval_Library$bloc_return	2	2	100.00 %	0	0	-
Eval_Library$circuit_return	1	2	50.00 %	0	0	-
Eval_Library$expr_return	2	2	100.00 %	0	0	-
Eval_Library$expr_scope	1	1	100.00 %	0	0	-
Eval_Library$function_return	2	2	100.00 %	0	0	-
Eval_Library$function_scope	1	1	100.00 %	0	0	-
Eval_Library$inparam_return	2	2	100.00 %	0	0	-
Eval_Library$inparam_scope	1	1	100.00 %	0	0	-
Eval_Library$ioparam_return	0	2	0.00 %	0	0	-
Eval_Library$ioparam_scope	0	1	0.00 %	0	0	-
Eval_Library$library_return	2	2	100.00 %	0	0	-
Eval_Library$library_scope	1	1	100.00 %	0	0	-
Eval_Library$list_return	2	2	100.00 %	0	0	-
Eval_Library$model_eq_return	2	2	100.00 %	0	0	-
Eval_Library$model_eq_scope	1	1	100.00 %	0	0	-
Eval_Library$model_return	2	2	100.00 %	0	0	-
Eval_Library$model_scope	1	1	100.00 %	0	0	-

Figure B.1 La couverture pour chacune des classes

Name	Lines	Total	%	Branches	Total	%
Eval_Library$outparam_return	2	2	100.00 %	0	0	-
Eval_Library$outparam_scope	1	1	100.00 %	0	0	-
Eval_Library$param_seq_return	2	2	100.00 %	0	0	-
Eval_Library$param_tri_return	0	2	0.00 %	0	0	-
Eval_Library$path_return	2	2	100.00 %	0	0	-
Eval_Library$path_scope	1	1	100.00 %	0	0	-
Eval_Library$piecewise_return	2	2	100.00 %	0	0	-
Eval_Library$pre_def_return	2	2	100.00 %	0	0	-
Eval_Library$pwl_list_return	2	2	100.00 %	0	0	-
Eval_Library$pwl_list_scope	1	1	100.00 %	0	0	-
Eval_Library$pwl_model_return	2	2	100.00 %	0	0	-
Eval_Library$pwl_model_scope	1	1	100.00 %	0	0	-
Eval_Library$stmt_return	2	2	100.00 %	0	0	-
Eval_Library$stmt_scope	1	1	100.00 %	0	0	-
Eval_Library$user_def_return	2	2	100.00 %	0	0	-
Eval_Library$user_def_scope	1	1	100.00 %	0	0	-
Eval_Logic	827	976	84.73 %	106	141	75.18 %
Eval_Logic$DFA14	11	12	91.67 %	0	0	-
Eval_Logic$bloc_return	2	2	100.00 %	0	0	-
Eval_Logic$circuit_return	1	2	50.00 %	0	0	-
Eval_Logic$expr_return	2	2	100.00 %	0	0	-
Eval_Logic$expr_scope	1	1	100.00 %	0	0	-
Eval_Logic$function_return	2	2	100.00 %	0	0	-
Eval_Logic$function_scope	1	1	100.00 %	0	0	-
Eval_Logic$inparam_return	2	2	100.00 %	0	0	-
Eval_Logic$inparam_scope	1	1	100.00 %	0	0	-
Eval_Logic$ioparam_return	2	2	100.00 %	0	0	-
Eval_Logic$ioparam_scope	1	1	100.00 %	0	0	-
Eval_Logic$library_return	2	2	100.00 %	0	0	-
Eval_Logic$library_scope	1	1	100.00 %	0	0	-
Eval_Logic$outparam_return	2	2	100.00 %	0	0	-
Eval_Logic$outparam_scope	1	1	100.00 %	0	0	-
Eval_Logic$param_seq_return	2	2	100.00 %	0	0	-
Eval_Logic$param_tri_return	0	2	0.00 %	0	0	-
Eval_Logic$pre_def_return	2	2	100.00 %	0	0	-
Eval_Logic$stmt_return	2	2	100.00 %	0	0	-
Eval_Logic$stmt_scope	1	1	100.00 %	0	0	-
Eval_Logic$user_def_return	2	2	100.00 %	0	0	-
Eval_Logic$user_def_scope	1	1	100.00 %	0	0	-
FunctionCell	45	46	97.83 %	10	10	100.00 %

Figure B.2 La couverture pour chacune des classes (suite)

Name	Lines	Total	%	Branches	Total	%
Instance	65	81	80.25 %	20	28	71.43 %
InstanceAnnotation	14	14	100.00 %	0	0	-
InstanceCell	139	172	80.81 %	54	72	75.00 %
LibraryLexer	472	591	79.86 %	134	207	64.73 %
LibraryLexer$DFA13	11	12	91.67 %	0	0	-
LibraryLexer$DFA4	11	12	91.67 %	0	0	-
LibraryParser	1612	1978	81.50 %	248	394	62.94 %
LibraryParser$DFA14	11	12	91.67 %	0	0	-
LibraryParser$addop_return	2	2	100.00 %	0	0	-
LibraryParser$atom_return	2	2	100.00 %	0	0	-
LibraryParser$bloc_return	2	2	100.00 %	0	0	-
LibraryParser$circuit_return	2	2	100.00 %	0	0	-
LibraryParser$expr_return	2	2	100.00 %	0	0	-
LibraryParser$function_return	2	2	100.00 %	0	0	-
LibraryParser$inparam_return	2	2	100.00 %	0	0	-
LibraryParser$ioparam_return	0	2	0.00 %	0	0	-
LibraryParser$library_return	2	2	100.00 %	0	0	-
LibraryParser$list_return	2	2	100.00 %	0	0	-
LibraryParser$rmodel_eq_return	2	2	100.00 %	0	0	-
LibraryParser$rmodel_return	2	2	100.00 %	0	0	-
LibraryParser$multExpr_return	2	2	100.00 %	0	0	-
LibraryParser$multop_return	2	2	100.00 %	0	0	-
LibraryParser$outparam_return	2	2	100.00 %	0	0	-
LibraryParser$param_seq_return	2	2	100.00 %	0	0	-
LibraryParser$param_tri_return	0	2	0.00 %	0	0	-
LibraryParser$path_return	2	2	100.00 %	0	0	-
LibraryParser$phase_return	2	2	100.00 %	0	0	-
LibraryParser$piecewise_return	2	2	100.00 %	0	0	-
LibraryParser$pre_def_return	2	2	100.00 %	0	0	-
LibraryParser$pwl_list_return	2	2	100.00 %	0	0	-
LibraryParser$pwl_model_return	2	2	100.00 %	0	0	-
LibraryParser$stmt_return	2	2	100.00 %	0	0	-
LibraryParser$user_def_return	2	2	100.00 %	0	0	-
LogicArgsParser	57	73	78.08 %	3	12	25.00 %
LogicLexer	375	460	81.52 %	121	191	63.35 %
LogicLexer$DFA13	11	12	91.67 %	0	0	-
LogicLexer$DFA3	11	12	91.67 %	0	0	-
LogicMain	15	23	65.22 %	0	4	0.00 %

Figure B.3 La couverture pour chacune des classes (suite)

Name	Lines	Total	%	Branches	Total	%
Instance	65	81	80.25 %	20	28	71.43 %
InstanceAnnotation	14	14	100.00 %	0	0	-
InstanceCell	139	172	80.81 %	54	72	75.00 %
LibraryLexer	472	591	79.86 %	134	207	64.73 %
LibraryLexer$DFA13	11	12	91.67 %	0	0	-
LibraryLexer$DFA4	11	12	91.67 %	0	0	-
LibraryParser	1612	1978	81.50 %	248	394	62.94 %
LibraryParser$DFA14	11	12	91.67 %	0	0	-
LibraryParser$addop_return	2	2	100.00 %	0	0	-
LibraryParser$atom_return	2	2	100.00 %	0	0	-
LibraryParser$bloc_return	2	2	100.00 %	0	0	-
LibraryParser$circuit_return	2	2	100.00 %	0	0	-
LibraryParser$expr_return	2	2	100.00 %	0	0	-
LibraryParser$function_return	2	2	100.00 %	0	0	-
LibraryParser$inparam_return	2	2	100.00 %	0	0	-
LibraryParser$ioparam_return	0	2	0.00 %	0	0	-
LibraryParser$library_return	2	2	100.00 %	0	0	-
LibraryParser$list_return	2	2	100.00 %	0	0	-
LibraryParser$model_eq_return	2	2	100.00 %	0	0	-
LibraryParser$model_return	2	2	100.00 %	0	0	-
LibraryParser$multExpr_return	2	2	100.00 %	0	0	-
LibraryParser$multop_return	2	2	100.00 %	0	0	-
LibraryParser$outparam_return	2	2	100.00 %	0	0	-
LibraryParser$param_seq_return	2	2	100.00 %	0	0	-
LibraryParser$param_tri_return	0	2	0.00 %	0	0	-
LibraryParser$path_return	2	2	100.00 %	0	0	-
LibraryParser$phase_return	2	2	100.00 %	0	0	-
LibraryParser$piecewise_return	2	2	100.00 %	0	0	-
LibraryParser$pre_def_return	2	2	100.00 %	0	0	-
LibraryParser$pwl_list_return	2	2	100.00 %	0	0	-
LibraryParser$pwl_model_return	2	2	100.00 %	0	0	-
LibraryParser$stmt_return	2	2	100.00 %	0	0	-
LibraryParser$user_def_return	2	2	100.00 %	0	0	-
LogicArgsParser	57	73	78.08 %	3	12	25.00 %
LogicLexer	375	460	81.52 %	121	191	63.35 %
LogicLexer$DFA13	11	12	91.67 %	0	0	-
LogicLexer$DFA3	11	12	91.67 %	0	0	-
LogicMain	15	23	65.22 %	0	4	0.00 %

Figure B.4 La couverture pour chacune des classes (suite)

LISTE DES RÉFÉRENCES

Blaauw, D., Rao, R., Srivastava, A. et Sylvester, D. (2004). Statistical analysis of subthreshold leakage current for VLSI circuits. *IEEE Transactions on VLSI Systems*, volume 12, numéro 2, p. 131–139.

Bohr, M. (2007). 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society News*, volume 12, numéro 1, p. 11–13.

Bozorgzadeh, E., Ghiasi, S., Sarrafzadeh, M. et Takahashi, A. (2003). Optimal integer delay budgeting on directed acyclic graphs. Dans *DAC '03 : Proceedings of the 40th Annual Design Automation Conference*. ACM, p. 920–925.

Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A. L., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S. A., Khatri, S. P., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G. et Villa, T. (1996). VIS : A system for verification and synthesis. Dans *Proceedings of the 8th International Conference on Computer Aided Verification*. Springer-Verlag, p. 428–432.

Bryant, R., Cheng, K., Kahng, A., Keutzer, K., Maly, W., Newton, R., Pileggi, L., Rabaey, J. et Sangiovanni-Vincentelli, A. (2001). Limitations and challenges of computer-aided design technology for CMOS VLSI. *Proceedings of the IEEE*, volume 89, numéro 3, p. 341–365.

Chen, C., Bozorgzadeh, E., Srivastava, A. et Sarrafzadeh, M. (2002). Budget management with applications. *Algorithmica*, volume 34, numéro 3, p. 261–275.

Chen, C., Yang, X. et Sarrafzadeh, M. (2000). Potential slack : An effective metric of combinational circuit performance. Dans *ICCAD '00 : Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, p. 202–207.

Cong, J. (1997). *Challenges and Opportunities for Design Innovations in Nanometer Technologies* (Rapport technique). Université de Californie, Los Angeles, 15 p.

Cormen, T. H., Stein, C., Rivest, R. L. et Leiserson, C. E. (2001). *Introduction to Algorithms*, 2e édition. McGraw-Hill Higher Education, 1184 p.

De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 576 p.

Dennard, R., Gaensslen, F. H., Yu, H., Rideout, V. L., Bassous, E. et Leblanc, A. R. (1999). Design of ion-implanted MOSFET's with very small physical dimensions. *Proceedings of the IEEE*, volume 87, numéro 4, p. 668–678.

Devadas, S., Ghosh, A. et Keutzer, K. (1994). *Logic Synthesis*. McGraw-Hill, 404 p.

Felt, E., Charbon, E., Malavasi, E. et Sangiovanni-Vincentelli, A. (1992). An efficient methodology for symbolic compaction of analog IC's with multiple symmetry constraints. Dans *EURO-DAC '92 : Proceedings of the Conference on European Design Automation*. IEEE Computer Society Press, p. 148–153.

Freeman, E., Freeman, E., Bates, B. et Sierra, K. (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc., 688 p.

Gamma, E., Helm, R., Johnson, R. E. et Vlissides, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 416 p.

Gao, T., Vaidya, P. M. et Liu, C. L. (1991). A new performance driven placement algorithm. Dans *ICCAD '91 : Proceedings of the 1991 International Conference on Computer-Aided Design*. IEEE Computer Society, p. 44–47.

Gács, P. et Lovász, L. (1981). Khachiyan's algorithm for linear programming. Dans *Mathematical Programming at Oberwolfach*, chapitre 5, *volume 14*. Springer Berlin Heidelberg, p. 61–68.

Ghiasi, S. (2005). Efficient implementation selection via time budgeting : complexity analysis and leakage optimization case study. Dans *ICCD '05 : Proceedings of the 2005 International Conference on Computer Design*. IEEE Computer Society, p. 127–129.

Ghiasi, S., Bozorgzadeh, E., Choudhuri, S. et Sarrafzadeh, M. (2004). A unified theory of timing budget management. Dans *ICCAD '04 : Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, p. 653–659.

Ghiasi, S., Huang, P. et Jafari, R. (2006). Probabilistic delay budget assignment for synthesis of soft real-time applications. *IEEE Transactions on VLSI Systems*, volume 14, numéro 8, p. 843–853.

Girard, P., Landrault, C., Pravossoudovitch, S. et Severac, D. (1997). A gate resizing technique for high reduction in power consumption. Dans *ISLPED '97 : Proceedings of the 1997 International Symposium on Low Power Electronics and Design*. ACM, p. 281–286.

Gosti, W., Narayan, A., Brayton, R. K. et Sangiovanni-Vincentelli, A. L. (1998). Wire-planning in logic synthesis. Dans *ICCAD '98 : Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*. ACM, p. 26–33.

Henderson-Sellers, B. (1996). *Object-oriented metrics : measures of complexity*. Prentice-Hall, Inc., 252 p.

Ho, R., Mai, K. W. et Horowitz, M. A. (2001). The future of wires. *Proceedings of the IEEE*, volume 89, numéro 4, p. 490–504.

Hurst, S. L., Miller, D. M. et Muzio, J. C. (1985). *Spectral Techniques in Digital Logic*. Academic Press, 314 p.

Hwang, M.-E., Jung, S.-O. et Roy, K. (2009). Slope interconnect effort : Gate-interconnect interdependent delay modeling for early CMOS circuit simulation. *Trans. Cir. Sys. Part I*, volume 56, numéro 7, p. 1427–1440.

Intel (2005). *Excerpts from* A conversation with Gordon Moore : Moore's law. Dans Intel, *PRESS KIT - Moore's Law 40th Anniversary*. `http:`

//download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf (page consultée le 24 mars 2010).

Jinks, P. (mars 2004). *BNF/EBNF variants.* http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html (page consultée le 11 mars 2011).

Kahng, A. B., Mantik, S. et Markov, I. L. (2002). Min-max placement for large-scale timing optimization. Dans *ISPD '02 : Proceedings of the 2002 International Symposium on Physical Design.* ACM, p. 143–148.

Keutzer, K., Newton, A. R. et Shenoy, N. (1997). The future of logic synthesis and physical design in deep-submicron process geometries. Dans *ISPD '97 : Proceedings of the 1997 International Symposium on Physical Design.* ACM, p. 218–224.

Kuo, C. C. et Wu, A. C. H. (2000). Delay budgeting for a timing-closure-driven design method. Dans *ICCAD '00 : Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design.* IEEE Press, p. 202–207.

Kursun, E., Ghiasi, S. et Sarrafzadeh, M. (2004). Transistor level budgeting for power optimization. Dans *ISQED '04 : Proceedings of the 5th International Symposium on Quality Electronic Design.* IEEE Computer Society, p. 116–121.

Lee, J. F. et Tang, D. T. (1987). VLSI layout compaction with grid and mixed constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 6, numéro 5, p. 903–910.

Leiserson, C. et Saxe, J. (1991). Retiming synchronous circuitry. *Algorithmica*, volume 6, numéro 1, p. 5–35.

Liao, Y. et Wong, C. K. (1983). An algorithm to compact a VLSI symbolic layout with mixed constraints. Dans *DAC '83 : Proceedings of the 20th Design Automation Conference.* IEEE Press, p. 107–112.

Lin, H. R. et Hwang, T. (1995). Power reduction by gate sizing with path-oriented slack calculation. Dans *ASP-DAC '95 : Proceedings of the 1995 Asia and South Pacific Design Automation Conference.* ACM, p. 7–12.

Luber, J. (mai 2008). *Quick Starter on Parser Grammars - No Past Experience Required.* http://www.antlr.org/wiki/display/ANTLR3/Quick+Starter+on+Parser+Grammars+-+No+Past+Experience+Required (page consultée le 11 mars 2011).

Mailhot, F. et De Micheli, G. (1993). Algorithms for technology mapping based on binary decision diagrams and on boolean operations. Dans *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* IEEE, p. 599–620.

Manohararajah, V., Chiu, G. R., Singh, D. P. et Brown, S. D. (2006). Difficulty of predicting interconnect delay in a timing driven FPGA CAD flow. Dans *Proceedings of the 2006 international workshop on System-level interconnect prediction.* ACM, p. 3–8.

Nair, R., Berman, C. L., Hauge, P. S. et Yoffa, E. J. (1989). Generation of performance constraints for layouts. *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 8, numéro 8, p. 860–874.

Otten, R. H. J. M. et Brayton, R. K. (1998). Planning for performance. Dans *DAC '98 : Proceedings of the 35th annual Design Automation Conference*. ACM, p. 122–127.

Parr, T. (2007). *The Definitive ANTLR Reference : Building Domain-Specific Languages*. Pragmatic Bookshelf, 376 p.

Rabbani, A. H. (2007). *Optimisation temporelle de circuits logiques par l'utilisation de tampons*. Mémoire de maîtrise, Université de Sherbrooke, Faculté de génie, Sherbrooke, Canada, 119 p.

Sarrafzadeh, M., Knol, D. A. et Tellez, G. E. (1997). A delay budgeting algorithm ensuring maximum flexibility in placement. Dans *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. IEEE Computer Society, p. 1332–1341.

Saxena, P. (2009). The evolution of interconnect management in physical synthesis. Dans *VLSI-DAT '09 : International Symposium on VLSI Design, Automation and Test*. ACM, p. 27–30.

Shenoy, N. et Rudell, R. (1994). Efficient implementation of retiming. Dans *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, p. 226–233.

Shilov, A. (juillet 2009). *Xbit Laboratories*. `http://www.xbitlabs.com/news/video/display/20090730072951_Nvidia_Chief_Scientist_11nm_Graphics_Chips_with_5000_Stream_Processors_Due_in_2015.html` (page consultée le 06 avril 2011).

Singhal, L., Bozorgzadeh, E. et Eppstein, D. (2007). Interconnect criticality-driven delay relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 26, numéro 10, p. 1803–1817.

Srivastava, A., Memik, S. O., Choi, B. et Sarrafzadeh, M. (2003). Achieving design closure through delay relaxation parameter. Dans *ICCAD '03 : Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*. ACM, p. 53–57.

Sutherland, I., Sproull, B. et Harris, D. (1999). *Logical Effort : Designing Fast* CMOS *Circuits*. Morgan Kaufmann Publishers Inc., 256 p.

Tellez, G. E., Sarrafzadeh, M. et Knol, D. (1997). Unification of budgeting and placement. Dans *DAC '97 : Proceedings of the 34th annual Design Automation Conference*. ACM, p. 758–761.

Yeh, C. et Marek-Sadowska, M. (2003). Delay budgeting in sequential circuit with application on FPGA placement. Dans *DAC '03 : Proceedings of the 40th annual Design Automation Conference*. ACM, p. 202–207.

Youssef, H. et Shragowitz, E. (1990). Timing constraints for correct performance. Dans *Proceedings of the International Conference on Computer-Aided Design*. IEEE Computer Society, p. 24–27.