Amélioration du processus de testabilité des circuits intégrés asynchrones dérivés de la topologie de conception d'Octasic

par

Quentin LAMBERT

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE ÉLECTRIQUE
M. Sc. A.

MONTRÉAL, LE 12 SEPTEMBRE 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Claude Thibeault, directeur de mémoire
Département de génie électrique à l'École de technologie supérieure

M. Dominic Deslandes, président du jury
Département de génie électrique à l'École de technologie supérieure

M. Pascal Giard, membre du jury
Département de génie électrique à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 27 AOÛT 2019

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je tiens en premier lieu à remercier mon directeur de mémoire Claude Thibeault de m'avoir orienté dans mes recherches pendant toute la durée de ma maitrise. Ses conseils et son expérience ont été indispensables à la réalisation de ce mémoire. Je veux aussi le remercier pour sa lecture détaillée et précise de ce document ainsi que pour les modifications qu'il a proposées. Enfin, je tiens aussi à souligner la contribution financière obtenue par monsieur Thibeault sans laquelle la poursuite de mes études aurait été impossible.

Je remercie également Mickaël Fiorentino, doctorant à l'école de Polytechnique Montréal, pour m'avoir partagé ses travaux de recherche qui ont largement contribué au développement des outils créés pour ce travail. Je le remercie aussi pour tout le soutien technique et moral qu'il m'a apporté durant ce mémoire.

Je veux aussi remercier Omar Al-Terkawi Hasib, doctorant à l'école de Polytechnique Montréal, pour m'avoir expliqué le fonctionnement des scripts fournis par Octasic ainsi que pour toutes les questions auxquelles il a pu répondre durant mes travaux de recherche.

Mes remerciements s'adressent aussi à Octasic et Daniel Crépeau pour m'avoir permis de consulter leurs travaux et m'ont donné l'inspiration pour le développement des procédures de test nécessaire a la génération automatisée des vecteurs de test.

Amélioration du processus de testabilité des circuits intégrés asynchrones dérivés de la topologie de conception d'Octasic

Quentin LAMBERT

RÉSUMÉ

Les circuits asynchrones regroupent une grande variété de technique de conception. Octasic, une entreprise montréalaise, conçoit des processeurs dédiés au traitement de signal (DSP) qui exploitent une solution asynchrone particulière de sa propre invention. La testabilité de ce type de circuit a déjà fait l'objet de précédents travaux de recherche qui ont mené au développement d'une première méthode de test et d'algorithme d'analyse. Cependant, l'aspect de l'automatisation de cette méthode qui est pourtant nécessaire à sa démocratisation n'a pour l'instant pas été traité.

Ce mémoire propose donc d'améliorer et d'automatiser autant que possible la méthode de test précédemment développée pour les circuits asynchrones qui utilisent la topologie de design des DSP d'Octasic. Contrairement à la méthode de test initial, le processus développé prend en charge la testabilité des machines à états finis et la gestion des structures de réseau d'horloges complexe qui peuvent contenir des registres. À travers plusieurs circuits asynchrones de différentes complexités, on introduit un flot de testabilité en partie automatisé qui débute à partir de la synthèse et se déroule jusqu'à la simulation des vecteurs de tests. De plus, on y présente un programme capable d'analyser l'arrangement interne des circuits pour intégrer et connecter les structures spécifiques à la technique de test. Enfin, on y expose le processus de création des procédures de test nécessaire à la génération automatisée des vecteurs de test.

Pour mesurer l'efficacité du flot de testabilité créé, des vecteurs de tests sont générés et simulés. Les tests menés grâce à l'outil d'ATPG *(Automatic Test Pattern Generator)* et la simulation des vecteurs de test nous permettent d'obtenir un taux de couverture de pannes de 76.08%. Ces tests exploitent la technique du *launch-on-capture* à vitesse nominale sur notre circuit le plus complexe, un microprocesseur mini-MIPS asynchrone dérivé de l'architecture des DSP d'Octasic et implémenté grâce à la technologie 45nm de Cadence.

Mots-clés : testabilité, circuits asynchrones, test de transition à vitesse nominale, automatisation

An improvement of testability in asynchronous integrated circuits derived from the design topology of Octasic

Quentin LAMBERT

ABSTRACT

Asynchronous circuits include a wide variety of design techniques. Octasic, a Montreal - based company, designs digital signal processors (DSPs) that are using a particular asynchronous technique of its own invention. This type of circuit has already been the subject of previous research that led to the development of an initial test method and analysis algorithm. Yet, automation of a method is often needed to be democratized, and this technique has never been dealt with.

Hence, this thesis proposes to improve and automate as much as possible the previously-developed test method for asynchronous circuits that are using Octasic's DSP structure. Unlike the initial method, the developed process supports testability of finite state machines and the management of complex clock networks that can contain registers. Across a variety of asynchronous circuits with variable complexity, we introduce a mostly-automated testability flow that starts from the synthesis of the design and runs until the test-vector simulation. In addition, it presents a program that can analyze internal structural arrangement of circuits to insert and connect specific modules of the test method. Finally, we explain how to write test procedures that are necessary to generate the test vectors with the automatic test-pattern generator (ATPG).

We generate test vectors and simulate them to measure the efficiency of our test method. The tests conducted with the ATPG tool and the simulation of the tests vectors allow us to obtain a test coverage rate of 76.08%. These tests exploit the launch-on-capture technique at nominal speed on our most complex circuit, an asynchronous mini-MIPS microprocessor derived from Octasic's DSP architecture and implemented with the 45nm Cadence technology.

Keywords: testability, asynchronous circuits, transition test at nominal speed, automation

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ALGORITHMES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ALU	Arithmetic logic unit
ASIC	Application-specific integrated circuit
ACK	Acknowledgement
ATPG	Automatic test-pattern generation
BIST	Built-in self test
BLIF	Berkeley logic interchange format
CAO	Conception assistée par ordinateur
CD	Converging datapath
CDL	Configurable delay line
CLC	Configurable logic cloud
DSP	Digital signal processor
DI	Delay insensible
DMEM	Direct memory
EU	Execution unit
FPGA	Field-programmable gate array
FSM	Finite state machine
ICC1D	Indirectly-converging category 1 datapath
ICC2D	Indirectly-converging category 2 datapath
ICCS1D	Indirectly-converging category special 1 datapath
LOC	*Launch-on-capture*
LOS	*Launch-on-shift*
LSSD	Level-sensitive scan design

NCL	Null conventiomnal logic
PC	Program counter
QDI	Quasi delay insensitive
RTL	Register transfer level
SCD	Special converging datapath
SE	Scan enable
SI	Speed independent
SI	Scan input
SO	Scan output
STF	Slow to fall
STR	Slow to rise
TCL	Tool command language
TDF	Transition-delay fault
TPG	Test-pattern generator
RA	Response analyzer
SNAP	Stanford network analysis platform
VHDL	Very-high-speed integrated circuit hardware-description language
YOSYS	Yosys synthesis suite

INTRODUCTION

Avec l'apparition d'une variété d'appareils mobiles toujours plus complexes depuis les années 2000 et pour des soucis d'autonomie, la demande de circuits intégrés basse consommation et de meilleures batteries se fait de plus en plus grande. Or, comme il ne devrait plus y avoir d'augmentation notable de la densité énergétique de ces dernières dans les années à venir (Hodson, 2015), il revient donc aux concepteurs de circuits la lourde charge de réduire la dépense énergétique des puces pour pouvoir augmenter l'autonomie des appareils mobiles. La majorité des circuits actuels sont synchrones; or, on sait que la majeure partie de la consommation d'énergie vient de l'horloge (Shinde & Salankar, 2011). Certaines techniques ont déjà été utilisées pour diminuer ou supprimer le signal d'horloge dans certaines parties du circuit, mais les fonctions de contrôle qui permettent ces prouesses consomment elles-mêmes de l'énergie.

Une solution plus radicale pour réduire la dépense énergétique serait de tout simplement supprimer l'horloge globale du circuit. C'est l'idée générale de la méthode de design asynchrone : au lieu que le circuit soit contrôlé de manière globale par une horloge, il est synchronisé localement par des échanges ponctuels entre diverses parties de la puce. Cette méthode de design est déjà utilisée en partie dans les techniques GALS (Globally Asynchronous Locally Synchronous) (Marc Renaudin, 2013) où une multitude de parties synchrones de la puce communique de manière asynchrone.

Toutefois, les conceptions purement asynchrones restent cantonnées à de rares applications qui nécessitent une consommation d'énergie ou une génération de bruit très faible, comme le domaine de la cryptographie par exemple (Marc Renaudin, 2013). Les raisons de cette timide adoption par l'industrie sont multiples. Tout d'abord, les langages de description matérielle comme le VHDL ou le Verilog sont peu adaptés à la conception asynchrone (Spars & Furber, 2002). Encore plus embêtants, les principaux outils de conception de circuit sont globalement tous focalisés sur la méthodologie de design synchrone (Kondratyev & Lwin, 2002). Enfin le manque de procédure de testabilité qui permet de déterminer si la puce est fonctionnelle ou

non est un des freins majeurs à l'adoption massive de la méthode de conception asynchrone (Zeidler & Krstić, 2015).

La recherche sur les designs asynchrones est en constante évolution. Il existe donc une grande variété de circuits asynchrones. Certaines méthodes invoquent des éléments spécifiques comme les portes logiques C qui aident dans la synchronisation des signaux. Plusieurs techniques utilisent des méthodes de synchronisation basées sur la communication des éléments qui utilisent des protocoles vérifiant le bon envoi et la bonne réception des données. Enfin certains designs tentent de retarder une impulsion en utilisant des lignes à délais afin de déclencher successivement des opérations dans un pipeline. Les concepteurs peuvent aussi décider de combiner plusieurs techniques dans la même puce. Pour ces raisons et dans la dernière décennie, une multitude de méthodes de test pour les circuits asynchrones ont été développées. Malgré tout, la tâche se révèle toujours compliquée à cause des problèmes de contrôlabilité et d'observabilité qui découlent de tous ces différents designs (Zeidler & Krstić, 2015).

Dans l'article "Exploiting Built-In Delay Lines for Applying *Launch-on-capture* At-Speed Testing on Self-Timed Circuits" (Hasib et al., 2018), les auteurs dévoilent une méthode qui permet de tester à vitesse de fonctionnement réelle les circuits asynchrones développés par Octasic, une compagnie à la base de plusieurs générations de DSP asynchrones. Cependant, les structures de test utilisées par la méthode employée dans l'article ont été implémentées manuellement par les ingénieurs d'Octasic, car il n'existe à cette heure aucun outil capable de le faire de manière automatisée. Ce mémoire a donc pour objectif d'améliorer et d'automatiser le processus d'insertion de ces structures de test dans les designs asynchrones de style Octasic, à savoir à données groupées sans protocole (Hasib et al., 2018) d'accusé de réception (*single-rail bundled-data handshake-free*).

Cet objectif principal peut être divisé en 5 sous-objectifs qui sont :

- automatiser l'insertion des structures de test spécifique définies en partie dans (Hasib et al., 2018);

- trouver un moyen pour déterminer le nombre et la connexion des signaux de contrôle des multiplexeurs de test de la technique de test développé par (Hasib et al., 2018);
- créer une méthode pour appliquer un test de transition de type *launch-on-capture* sur les machines à états finis asynchrones;
- obtenir une couverture des pannes satisfaisante dans des circuits asynchrones de complexité différente à l'aide du flot de testabilité développé;
- analyser l'impact des structures de test introduites sur la surface du circuit.

Ces 5 sous-objectifs sont explorés à travers des 6 chapitres de ce mémoire. En plus de présenter les notions de base nécessaires à la compréhension du reste du mémoire, le CHAPITRE 1 fait l'état de l'art de la testabilité des circuits asynchrones. De plus, il présente plus en détail l'article de Hasib (Hasib et al., 2018) sur lequel est basé nos travaux. On y expose aussi succinctement les travaux préliminaires de TÊTU (TÊTU, 2014) sur la recherche des points de convergence qui est une étape nécessaire à la réalisation de notre 2ème sous-objectif. Le CHAPITRE 2, qu'on appelle banc d'essai, introduit tous les circuits asynchrones sur lesquels nous avons travaillé. On retrouvera ce banc d'essai dans les chapitres 3, 5 et 6. Le CHAPITRE 3 sur la stratégie de test a principalement deux objectifs : présenter les différentes structures de test utilisées dans ce mémoire et montrer leur intégration dans les circuits du banc d'essai. C'est aussi en partie dans ce chapitre qu'on répond au 3ème sous-objectif, c'est-à-dire le test de machines à états finis asynchrones. Le CHAPITRE 4 correspond à la plus grosse contribution apportée par ce mémoire. Il répond au sous-objectif 1 et 2 en décrivant l'automatisation de l'insertion des structures de test spécifique ainsi que la détermination du nombre et de la connexion des signaux de contrôle de test. Le CHAPITRE 5 répond au 4ème sous-objectif en s'attelant à maximiser la couverture de pannes des circuits du banc d'essai à travers la génération de vecteurs de test. Pour ce faire, on y décrit la stratégie des procédures de test employée pour les circuits du banc d'essai. On y présente aussi la description de ces différentes procédures, une étape nécessaire à la génération des vecteurs de test par l'ATPG. Pour finir, le CHAPITRE 6 présente et analyse tous les résultats produits par le flot de testabilité. On s'intéresse aux graphes et matrices produits par les algorithmes décrits dans le CHAPITRE 4, mais aussi à la couverture

de pannes des circuits générée par la stratégie décrite dans le CHAPITRE 5. De plus, ce chapitre répond au 5ème sous- objectif en analysant l'impact des structures de test insérées sur la surface des circuits du banc d'essai. On verra aussi dans ce chapitre pourquoi la détection de machine à états finis qui correspond à une partie du 1er objectif ne fonctionne pas. On clôture ce mémoire avec une conclusion générale et quelques recommandations pour des améliorations futures du flot automatisé développé.

CHAPITRE 1

NOTIONS DE BASE ET REVUE DE LITTÉRATURE

1.1 Introduction

Le but de ce chapitre est de couvrir l'état de l'art des travaux de recherche relatif à l'automatisation des processus de testabilité dans les circuits asynchrones. Dans une première section, on commencera par introduire les circuits asynchrones ainsi que quelques notions de base concernant le test de circuits intégrés. Puis, la revue de littérature sera divisée en deux sous-parties. Dans la première sous-partie, on résumera l'état de l'art de la testabilité des circuits asynchrones : les défis ainsi que les méthodes de test sur ce type de circuit seront détaillés. Nous finirons par résumer l'état des processus automatisés de testabilité des circuits asynchrones. La deuxième sous-partie sera l'occasion de présenter la topologie de circuit introduite par Octasic avec laquelle nous avons travaillé. La méthode de test dont nous avons eu la charge d'automatiser sera présentée. Nous finirons par exposer des travaux préliminaires déjà réalisés dans le cadre de l'automatisation de cette méthode de test.

1.2 Circuits asynchrones

Cette première partie a pour objectif de présenter les circuits asynchrones afin de fournir au lecteur quelques notions utiles à la compréhension de ce mémoire. La première partie « Classes » permettra de présenter les différents types de circuits asynchrones. Deux types de communication relatifs aux micropipelines seront présentés dans la section « Protocoles ». Enfin, dans « Codages », on introduira les différentes manières de coder les données.

1.2.1 Classes

Selon (Berthier, 2016) et (Rios, 2008), on compte 5 classes de circuit asynchrone : la logique insensible au délai (*DI : Delay Insensible*), la logique quasi-insensible au délai (*QDI : Quasi-Delay-Insensitive*), les circuits indépendants de la vitesse *(SI : Speed Independant),* les circuits de Huffman et les circuits micropipeline.

La logique insensible au délai n'impose aucune contrainte de temps sur le circuit. Ainsi, on peut utiliser n'importe quelle porte logique ou longueur de fil sans se soucier du délai qu'ils impliquent. Cette catégorie de circuit est très limitée et leur taille représente un désavantage important.

La logique quasi-insensible au délai reprend les mêmes hypothèses que celle insensible aux délais. La seule différence est que l'on suppose que certaines fourches (*fanout*) peuvent être isochrones. Dans une fourche isochrone, le délai dans une branche est le même dans toutes les autres branches (Figure 1.1) (Brégier, 2007). Ce modèle est utilisé par un grand nombre de processeurs asynchrones (Mickaël Fiorentino, 2017a) et type de logique comme les circuits de type NCL (*Null Conventionnal Logic*).

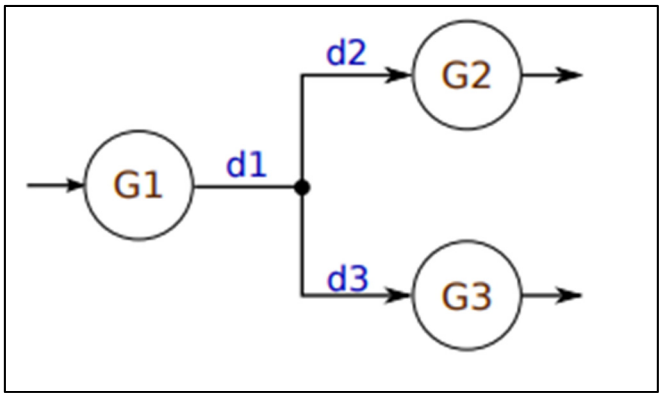

Figure 1.1 Le circuit est considéré comme quasi insensible aux délais si les délais d2 et d3 sont égaux, la fourche est considérée comme isochrone
Tirée de Brégier (2007, p. 12)

Les circuits indépendants de la vitesse sont similaires aux circuits QDI. La seule différence est que l'on considère toutes les fourches comme isochrones (Figure 1.1). Les circuits QDI sont davantage utilisés, car ils contiennent plus d'information et moins d'hypothèses temporelles que les circuits indépendants à la vitesse.

Les circuits de Huffman sont définis sous la forme d'une machine à états finis. Son évolution n'est pas cadencée par une horloge globale. On fixe plutôt un temps sur les délais du calcul de l'état futur ainsi que les entrées et sorties (Figure 1.2) (Brégier, 2007).

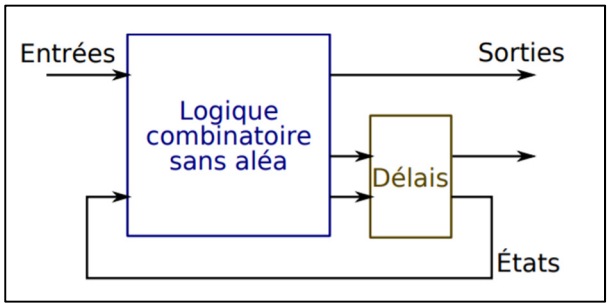

Figure 1.2 Structure d'un circuit de Huffman
Tirée de (Brégier, 2007)

Tous comme leur homologue synchrone, les circuits micropipelines sont composés de blocs de logique combinatoire interconnectés par des registres (Figure 1.3). L'horloge globale est remplacée par des modules qui permettent de synchroniser localement les registres. Cette synchronisation est arbitrée à l'aide de signaux de requête et d'acquittement selon divers protocoles.

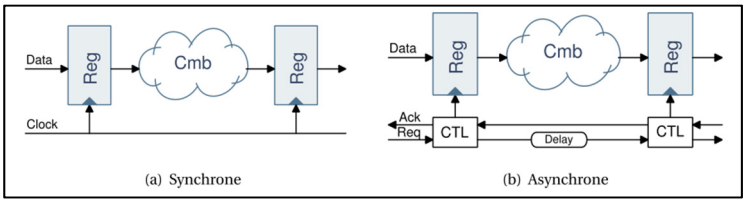

Figure 1.3 Comparaison d'un circuit synchrone et d'un circuit asynchrone micropipeline
Tirée de (Mickaël Fiorentino, 2017a)

Comme il sera décrit plus loin, les circuits ciblés dans ce mémoire, ceux de la compagnie Octasic, ressemblent à des micropipelines. Ils diffèrent cependant au niveau des signaux de requête et d'acquittement.

1.2.2 Protocoles

Il existe principalement deux protocoles notamment utilisés dans les circuits micropipelines. Ils utilisent le principe d'accusé de réception (*handshaking)*, selon un protocole à 4 phases ou à 2 phases.

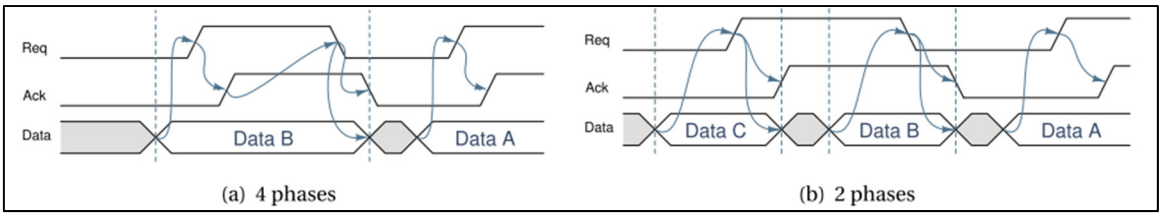

Figure 1.4 Protocole de communication asynchrone
Tirée de (Mickaël Fiorentino, 2017a)

Le protocole à 4 phases se déroule de la manière suivante (Figure 1.4 a):

1. L'émetteur pose la donnée sur la ligne de donnée et active le signal de requête *Req* à 1 afin de prévenir le récepteur.
2. Le récepteur s'acquitte de la bonne réception de la donnée et de la requête en activant le signal *Ack* à 1.
3. L'émetteur désactive la requête en passant le signal *Req* à 0 et invalide la donnée.
4. Le récepteur finalise la communication en remettant le signal d'acquittement à 0.

Par opposition au protocole à 4 phases, le protocole à 2 phases inverse l'état des signaux de requête et d'acquittement au lieu de leur faire correspondre des niveaux logiques fixes (Figure 1.4 b) :

1. L'émetteur pose la donnée sur la ligne de donnée et inverse le signal de requête afin de prévenir le récepteur.
2. Le récepteur s'acquitte de la bonne réception de la donnée et de la requête en inversant le signal *Ack*. La donnée sur le bus est invalidée et l'émetteur peut alors poser une autre donnée sur la ligne.

Comme il sera décrit plus loin, les circuits de la compagnie Octasic n'utilisent pas de protocoles.

1.2.3 Codages

Le codage des données peut se faire sur un ou plusieurs fils, mais deux méthodes sont principalement utilisées : le codage avec 2 fils (*Dual rail*) et le codage en données groupées (*Bundled data*) (Figure 1.5).

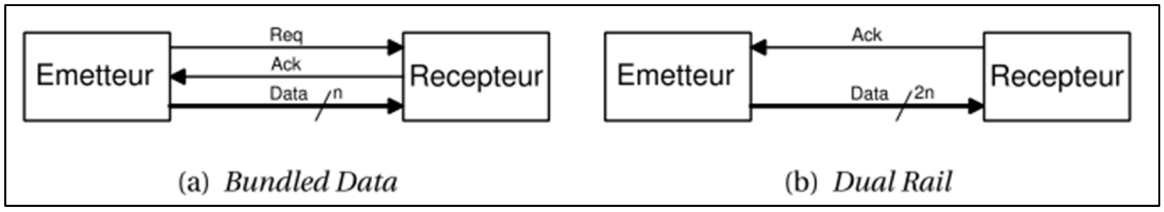

Figure 1.5 Différences entre le codage à données groupées (Bundle Data)
et le codage sur deux fils (Dual Rail)
Tirée de (Mickaël Fiorentino, 2017a)

Le codage *Dual Rail* intègre le signal de requête dans le bus de données. Chaque bit est codé sur deux fils, un fil code le niveau logique 0 tandis que l'autre fil code le niveau logique 1. Le signal de requête est implicite, car le récepteur accuse de la réception des données lorsqu'il a au moins reçu un niveau logique 1 sur l'un des bits. Cette détection est réalisée par un élément très souvent utilisé en logique asynchrone, la porte C.

Le codage de type *Bundled data* est le même dont nous avons l'habitude avec les techniques synchrones, c'est-à-dire que 1 bit est codé sur 1 fil. Les signaux de requête et d'acquittement

sont ajoutés à l'extérieur du bus de données, s'il y a lieu. Comme il sera décrit plus loin, les circuits de la compagnie Octasic utilisent le codage de type *Bundled data*.

1.3 Notions de base sur le test

Cette section porte sur les notions de base au niveau du test des circuits intégrés. On y aborde les thèmes suivants :

- le test fonctionnel;
- le test structurel;
- les indicateurs de mesure de qualité de test;
- les modèles de pannes;
- les chaînes de balayage;
- la technique de test « launch-on-shift »; et
- la technique de test « *launch-on-capture* ».

1.3.1 Test fonctionnel

Le test fonctionnel consiste à vérifier la bonne fonctionnalité du circuit à la sortie du procédé de fabrication. On teste le circuit en mode de fonctionnement normal. Ce type de test est non structuré, c'est-à-dire qu'il n'est basé sur aucune théorie ou technique particulière. En pratique, le test fonctionnel a été est très utilisé par les concepteurs et ingénieurs de test pour vérifier le circuit au plus proche de son fonctionnement nominal. Le test fonctionnel nécessite un testeur automatique très couteux, car le test est réalisé à vitesse nominale et demande donc au testeur d'être rapide et précis dans ses mesures. Les tests étant développés manuellement, le temps de développement induit indubitablement un retard de mise sur le marché de la puce et produit donc un coût supplémentaire pour l'entreprise (Maxwell, Hartanto, & Bentz, 2000). Enfin, le faible taux de couverture de panne et le manque d'indicateurs qui permettent de quantifier les erreurs ont incité les ingénieurs à passer au test structurel (Maxwell et al., 2000) sur lequel nous élaborerons dans la prochaine section

1.3.2 Test structurel

Tandis que le test fonctionnel vérifie la fonctionnalité du circuit, le test structurel se concentre à chercher si des pannes existent, i.e., des pannes induites par le procédé de fabrication. Ce type de test est très utilisé, car il est beaucoup plus rapide que le test fonctionnel. En effet, contrairement au test fonctionnel dont les séquences de test sont définies par l'homme, le test structurel utilise généralement un générateur automatique de vecteur de test (*ATPG : Automatic Test Pattern Generator*) associé à un modèle de panne (décrit plus loin) pour analyser le circuit et dresser la liste des pannes. Par conséquent, le test structurel diminue les coûts du processus et augmente la fiabilité du processus de testabilité. Des structures dédiées au test sont généralement insérées dans le circuit comme du test par balayage que nous allons voir plus loin. Notons que les générateurs automatiques de vecteur de test fournissent également des indicateurs de mesure de qualité de test, décrits dans ce qui suit.

1.3.3 Indicateur de mesure de qualité du test

Plusieurs indicateurs sont associés au test structurel. Ces indices permettent de mesurer la qualité du test pour maximiser le nombre de circuits adéquatement identifiés comme fonctionnels ou fautifs, à la fin de la production. On retrouve ces informations dans les rapports fournis par les outils de génération de vecteur de test, comme Tessent par exemple.

1.3.3.1 Taux de couverture de pannes

Le test structurel ne peut pas garantir la détection de toutes les pannes liées aux étapes de production du circuit. L'ATPG produit des vecteurs de tests qui visent un modèle de panne préalablement choisi. La capacité d'un groupe de vecteurs à détecter les pannes est donnée par le *Taux de couverture de panne* (Wang, Wu, & Wen, 2006).

$$Taux\ de\ couverture\ de\ panne = \frac{Nombre\ de\ pannes\ détectées}{Nombre\ total\ de\ pannes} \times 100\% \qquad (1.1)$$

1.3.3.2 Taux d'efficacité de la détection des pannes

Il est rarement possible de détecter toutes les pannes dans un circuit, certaines étant indétectables selon le modèle de panne choisi. On mesure alors le *taux d'efficacité de la détection des pannes* qui prend en compte les pannes indétectables (Wang et al., 2006).

$$Taux\ d'efficacité\ de\ la\ détection\ de\ pannes = \frac{Nombre\ de\ pannes\ détectées}{Nombre\ total\ de\ pannes - Nombre\ de\ pannes\ indétectables} \times 100\% \quad (1.2)$$

1.3.4 Modèles de panne

Selon (Wang et al., 2006), un modèle de panne est une représentation des défectuosités physiques qui conduisent le circuit à ne plus fonctionner de la manière attendue. On présente ici, le modèle de collage qui est le plus utilisé et le modèle de délai, car ce sont ceux que nous exploitons dans ce mémoire.

1.3.4.1 Modèle de collage

Les pannes de type collé-à (*stuck-at*) affectent l'état de la logique dans un circuit. Elles peuvent apparaître sur n'importe quel nœud interne et externe du circuit. Lorsqu'une panne de ce type se produit, le signal affecté se retrouve bloqué à un niveau logique précis. On dit que le signal est collé-à 0 ou collé-à 1 (Wang et al., 2006).

1.3.4.2 Modèle de délai

Une panne de délai est le résultat d'une augmentation du délai de propagation dans un chemin du circuit de telle sorte que le délai total dépasse la limite préalablement spécifiée. Or, la réduction de la finesse de gravure entraîne inévitablement un raccourcissement des délais internes des circuits intégrés. Par conséquent, les nouvelles puces sont de plus en plus

sensibles aux variations de délai induit par le procédé de fabrication (Verma, Kaushik, & Singh, 2010). Les pannes de délai sont donc de plus en plus présentes. Les modèles de délai se décomposent principalement en deux sous-parties, les pannes de transition et les pannes de chemin (Wang et al., 2006).

Une panne de chemin a lieu lorsque la somme des délais à travers les portes et les fils du chemin testé est supérieure à limite de délai spécifié entre deux éléments de mémorisation. En logique synchrone, cette limite de délai correspond à la période d'horloge si l'on néglige certains autres délais comme les temps de propagation et de maintien. En logique asynchrone, cela correspond au temps minimum pour que les deux registres qui entourent le chemin soient déclenchés successivement (Smith, 1985). La panne affecte l'ensemble du chemin.

Lorsque le résultat en sortie d'un changement de niveau logique à l'entrée d'une porte met plus de temps que prévu à se produire, on parle de panne de transition. L'effet de la panne est ponctuel puisqu'elle affecte le nœud connecté à la sortie de la porte défaillante. Ce type de pannes est divisé en deux sous-catégories, les transitions lentes à descendre STF (*Slow To Fall*) et les transitions lentes à monter STR (*Slow To Rise*) (Waicukauski, Lindbloom, Rosen, & Iyengar, 1987). Par exemple, une panne de type STF indique que le temps de la transition entre un niveau logique 1 et un niveau logique 0 est trop long et est susceptible de provoquer un problème de synchronisation.

Le modèle de panne *Transition Delay Fault* (TDF) utilise les pannes de transition.

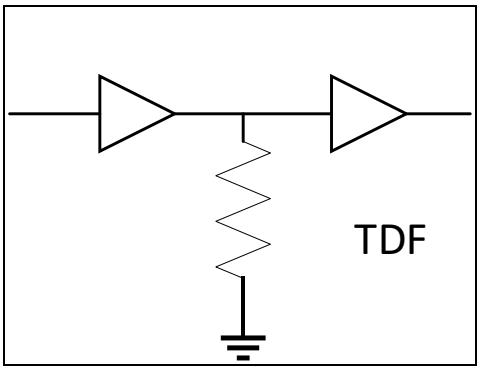

Figure 1.6 Modèle de pannes TDF

Le modèle de panne TDF est très utilisé. La panne est représentée soit par une résistance de rappel dans le cas STR, ou une résistance de tirage dans le cas STF. La technique est donc capable de différencier les pannes de type STR ou STF. La Figure 1.6 représente le cas d'une panne TDF de type STR ou la résistance de rappel ralentit la transition montante. Les logiciels utilisés dans ce mémoire sont configurés pour utiliser le modèle TDF.

1.3.5 Chaîne de balayage

Une des approches de conception facilitant le test structurel la plus utilisée est l'insertion d'une (ou plusieurs) chaîne(s) de registre à balayage. Cette insertion consiste à remplacer les registres normaux par des registres à balayage et à les relier entre eux afin de créer une (ou plusieurs) chaîne. Plusieurs types de registres à balayage existent. On étudie ici les registres à balayage qui utilisent un multiplexeur à l'entrée de donnée *D*, car c'est cette méthode qui est utilisée massivement dans l'article de sur lequel est basé ce mémoire.

Un registre à balayage de ce type est une bascule D classique avec un multiplexeur à son entrée *D* (Figure 1.7). Il permet de choisir entre l'entrée standard du fonctionnement normal du circuit et le signal *SI (Scan Input)* qui correspond à la connexion à la chaîne de balayage. La sélection de l'entrée est opérée par un signal *SE (Scan Enable)*.

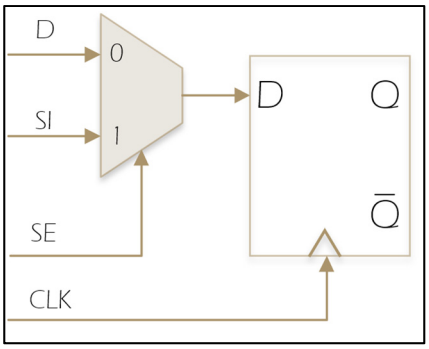

Figure 1.7 Registre à balayage

Une chaîne de balayage est formée de plusieurs registres à balayage en série. Ainsi le vecteur de test rentre par le signal *SI* du premier registre de la chaine et sort par le signal *SO (Scan Output)* branché au signal *Q* de la dernière bascule de la chaine (Figure 1.9). Le chargement d'un vecteur dans la chaîne de balayage permet d'initialiser tous les registres à une certaine valeur préalablement déterminée par le générateur automatique de vecteur de test.

L'insertion des registres à balayage ainsi que leurs connexions est généralement réalisée après la synthèse du design validé comme fonctionnel.

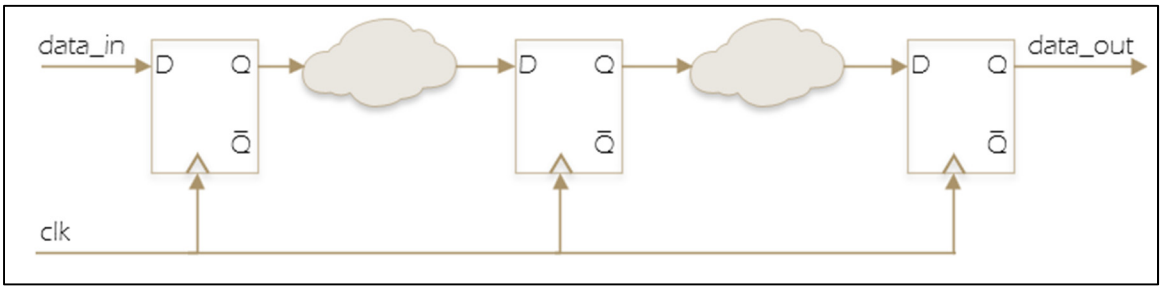

Figure 1.8 Pipeline synchrone sans structure de test

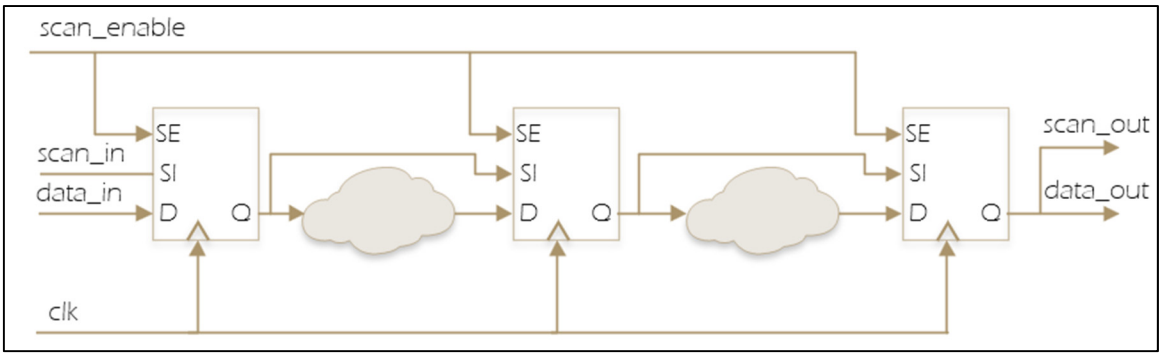

Figure 1.9 Pipeline synchrone avec insertion et connexion de la chaîne de balayage

La chaîne de balayage peut être exploitée pour exécuter des tests de transition TDF. Deux méthodes sont principalement utilisées, le *Launch-on-shift* et le *launch-on-capture*.

1.3.6 Launch-on-shift

La technique du launch-on-shift (LOS) ou skewed-load est inventée en 1993 par (Savir & Patil, 1993). Comme toutes les méthodes basées sur les registres à balayage, la première étape consiste à charger le vecteur de test dans les bascules. La particularité du test est que l'impulsion de lancement a lieu pendant le dernier décalage du vecteur. Le circuit est ensuite rapidement passé en mode de fonctionnement nominal pour que la 2e impulsion soit générée à vitesse nominale (*at-speed*). Comparativement au *launch-on-capture*, le LOS est plus facile à gérer par l'ATPG, car il n'a pas besoin de calculer les valeurs de transition à travers la logique combinatoire. De plus la couverture de panne est généralement plus élevée en utilisant le LOS. Malgré tout, le *launch-on-capture* reste le plus populaire. Une des difficultés d'implémentation du LOS est due au signal *scan_enable*. En effet, pour que le *scan_enable* change le mode des registres au même moment, le signal doit être implémenté comme une horloge, ce qui peut rajouter des difficultés supplémentaires lors du routage du circuit. Enfin, un autre problème important du LOS est le fait qu'il puisse tester des pannes qui ne soient pas possibles en mode de fonctionnement normal, augmentant alors artificiellement la couverture de panne.

1.3.7 *Launch-on-capture*

La méthode de test *launch-on-capture* (LOC) , aussi appelée *broad-side delay test* (Savir & Patil, 1994), nécessite plus de vecteurs de test et possède une couverture de panne plus faible que le *Launch-on-shift* . Comme le cycle de lancement est clairement séparé du cycle de décalage, la méthode n'impose pas d'exigence de vitesse sur le signal d'activation du mode de balayage *scan enable (SEN)* à la Figure 1.10. Par conséquent, la technique est utilisée dans les circuits à grande vitesse (Park & McCluskey, 2008). Comme décrit sur le chronogramme du LOC (Figure 1.10) après une première période de chargement du vecteur dans les registres à balayage, le signal S_{EN} est mis à 0. Puis, le lancement et la capture sont réalisés par deux impulsions successives sur le signal d'horloge. Enfin, S_{EN} est remis à 1 pour être prêt à lancer un nouveau test.

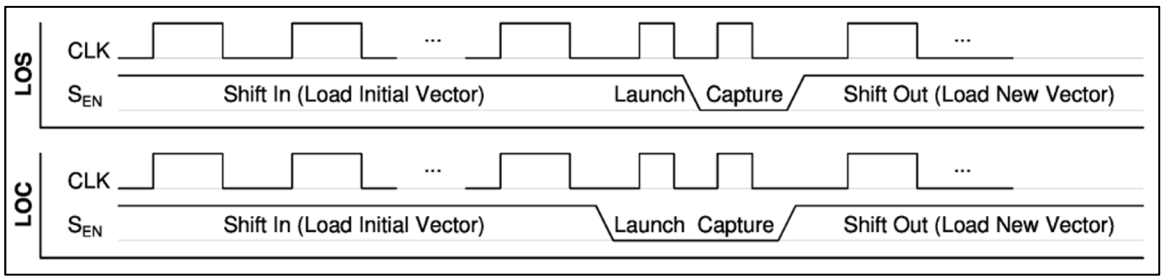

Figure 1.10 Comparaison des techniques de *Launch-on-shift* et *Launch-on-capture*
Tirée de (Hasib et al., 2018)

1.4 État de l'art

Cette section décrit dans un premier temps la testabilité des circuits asynchrones en général, les défis qui y sont associés, les principales méthodes de test utilisées ainsi que leur niveau d'automatisation. On y présente par la suite les circuits asynchrones de type Octasic, incluant le fonctionnement de la logique, la méthode de test de pannes de délai développée avant le début du projet lié à ce mémoire ainsi que les travaux préliminaires visant à faciliter son automatisation. Ce mémoire s'inscrit donc dans la suite de ces travaux.

1.4.1 Testabilité des circuits asynchrones

1.4.1.1 Les défis

Le monde numérique est dominé par la croissance des techniques de conception synchrones depuis maintenant près de cinquante ans (Venkat Satagopan, Bhaskaran, Al-Assadi, & Smith, 2005). Cette hégémonie de la technique synchrone a donc eu le temps de s'installer dans les habitudes des concepteurs et il est d'autant plus difficile de faire adopter les techniques asynchrones. De plus, bien que certaines solutions logicielles existent, le manque d'outils de conception assistée par ordinateur (CAO) et de maturité de ceux-ci (Zeidler & Krstić, 2015) n'encourage pas l'adoption des techniques asynchrones. C'est notamment le cas pour les outils de génération de vecteur de test automatique (ATPG) existant; le test des circuits asynchrones est un problème majeur par rapport aux circuits synchrones, car les ATPG ont été créés pour supporter les circuits basés sur une horloge globale (V. Satagopan, Bhaskaran, Al-Assadi, Smith, & Kakarla, 2007). Enfin, le manque de standardisation des circuits asynchrones est une des raisons pour laquelle peu de solutions spécifiques ont été créées.

Comme il a été décrit précédemment, il existe une grande variété de techniques de conception asynchrone. Tous ces circuits ont un point commun, par définition, il ne possède pas d'horloge globale. Cela pose alors un des défis majeurs de la testabilité de circuits asynchrones, c'est à dire le manque de contrôle durant le test (Zeidler & Krstić, 2015). En effet, tandis qu'un circuit synchrone peut être testé étape par étape en appliquant une impulsion d'horloge qui se propage alors à tous les registres du design, ce n'est pas possible dans un circuit asynchrone puisque celui-ci possède une multitude d'horloges locales, qui peuvent déclencher la propagation de données dans des structures complexes. Le circuit n'a alors pas d'état global, mais plutôt une multitude d'états localisés qui se synchronisent seulement lorsque le circuit a besoin de réaliser l'opération. Il en découle alors des problèmes de contrôlabilité, par exemple, la difficulté à mettre le circuit dans un état x qui aurait besoin d'être testé, et des problèmes d'observabilité, c'est à dire, la difficulté à vérifier dans quel état est le circuit (Te Beest, Peeters, Van Berkel, & Kerkhoff, 2003). Tel que décrit à la

section 3.2.3, c'est un problème que nous avons rencontré dans le mini-MIPS et qui nous a poussés à court-circuiter certaines structures lors du test pour éviter que l'exécution du test dépende de l'état du circuit.

Comme mentionné précédemment, lorsque l'on parcourt la littérature sur le test de circuits asynchrones, on se rend rapidement compte qu'il y a une grande variété de types de circuit et de méthodes de test associées. Contrairement au circuit synchrone, les tests ne peuvent pas être appliqués sans un effort substantiel de la part du concepteur. Tandis qu'il suffira d'exécuter quelques commandes sur des outils de test pour les circuits synchrones, la réalité est tout autre pour son pendant asynchrone. En effet, il faut d'abord commencer par choisir une méthode de test adaptée au circuit qui couvrira le plus de pannes possibles tout en n'ajoutant pas trop de nouvelle logique dédiée au test. Le concepteur devra ensuite modifier manuellement son design pour que celui-ci puisse exécuter la méthode de test préalablement choisi (Zeidler & Krstić, 2015). Or, pour qu'un ASIC soit implémenté avec succès sur silicium, un bon processus de design automatisé est indispensable (Venkat Satagopan et al., 2005) et c'est spécifiquement aux méthodes d'automatisation de la testabilité que nous nous intéressons dans ce mémoire.

1.4.1.2 Les méthodes de test

Les méthodes de test de circuit asynchrone sont nombreuses. D'après (Zeidler & Krstić, 2015), il existe quatre grandes techniques : le test fonctionnel, l'utilisation des propriétés du circuit asynchrone pour s'auto-évaluer, le BIST (*Built-In Self- Test*) et le test par chaîne de balayage.

Le test fonctionnel, décrit précédemment, est la première méthode à avoir été utilisée. Elle a pour avantage d'être non invasive et n'impacte donc pas la surface ni les performances du circuit. Elle a notamment été utilisée pour tester le microcontrôleur asynchrone 80C51 de chez Philips (Van Gageldonk et al., 1998). En plus des limitations vues dans la partie « Notions de base », le test fonctionnel possède d'autres inconvénients lorsque réalisé avec

des équipements de test dû au comportement non déterministe de la logique asynchrone. En effet, comme les équipements de test ont été conçus pour les circuits synchrones, ils s'attendent à recevoir la réponse à un test à un instant précis. Puisque le circuit asynchrone n'est par définition pas synchronisé sur la fréquence de l'équipement, la réponse au test pourrait arriver avant ou après la mesure de celui-ci.

D'autres méthodes de test utilisent la capacité des circuits asynchrones à effectuer les accusés de réception pour s'auto-évaluer. En effet, la présence d'une panne collée-à dans un des fils nécessaires aux protocoles d'accusé de réception peut arrêter le fonctionnement du circuit puisque celui-ci sera en attente d'une activation du signal de requête ou d'acquittement qui ne peut avoir lieu (Hulgaard, Burns, & Borriello, 1995). Cette particularité est notamment utilisée dans les travaux de (Piestrak & Nanya, 1995) et (David, Ginosar, & Yoeli, 1995).

Le BIST (*Built-In Self-Test*) est une autre manière courante de tester les circuits asynchrones. Comme son nom l'indique, le principe est d'intégrer une structure capable de tester le circuit directement dans celui-ci. Le BIST est souvent composé d'un générateur de patron de test TPG *(Test-Pattern Generator*) et d'un analyseur RA *(Response Analyzer*) qui analyse les données en sortie du circuit testé. L'avantage considérable de cette méthode provient du fait que le test peut être complètement personnalisé et adapté au circuit. Sa faiblesse réside souvent dans le manque de possibilité de diagnostic, car la réponse du test est compressée par le RA.

Enfin, la dernière méthode à décrire est celle qui a été le plus éprouvée et qui est la plus utilisée par l'industrie : le test par balayage (*scan-testing*). C'est cette grande famille de test qui nous intéresse le plus dans ce mémoire, car c'est celle qui est utilisé dans l'article sur lequel est basé ce mémoire. Deux types de pannes sont principalement analysées par ces techniques : les pannes collées-à et les pannes du au délai.

Philips présente en 2003 plusieurs méthodes (Te Beest et al., 2003; van Berkel, Peeters, & te Beest, 2003) qui permettent d'atteindre un taux de couverture de pannes de plus de 99% avec

le modèle collé-à. Le même laboratoire réduit de 49% à 58% par rapport à sa précédente solution la surface nécessaire pour implémenter le test avec une nouvelle méthode basée sur des multiplexeurs (F. te Beest & Peeters, 2005). En plus de l'insertion classique de verrou et registre à balayage, la technique consiste à insérer des multiplexeurs avec les portes C qui font partie de la boucle combinatoire. Ces multiplexeurs rendent les portes C testables en les reliant à la chaîne de balayage. Contrairement à leur précédente technique, cette solution a pour avantage d'utiliser uniquement des cellules standardisées. En effet, même la porte C est un assemblage de fonction logique de base (Figure 1.11). L'utilisation de cellule standard rend la méthode plus facile et économique à porter sur de nouvelle technologie. Cependant, selon (Sparsø, 2001), une telle utilisation peut faire en sorte que le circuit perde ses caractéristiques SI ou DI.

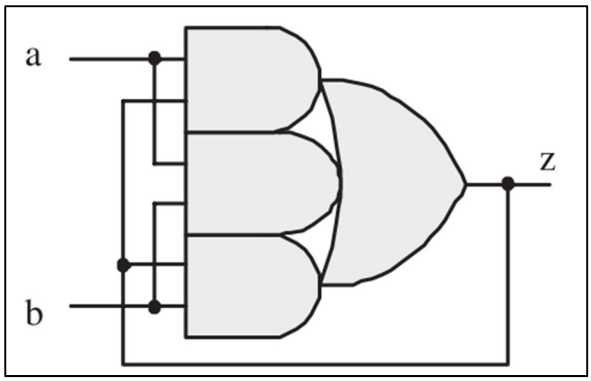

Figure 1.11 Implémentation d'une
porte C à deux entrées
Tirée de (F. te Beest & Peeters, 2005)

Petlin a conçu une méthode de test par balayage qui utilise le *Level Sensitive Scan Design* (LSSD), une solution qui utilise des pairs de verrou à la place de simple registre à balayage. La technique a été utilisée sur l'un des premiers processeurs asynchrones: l'AMULET1. La méthode permet aussi de détecter les pannes dues au délai (Petlin & Furber, 1995) .

Pour réduire davantage la surface des structures de test nécessaire, Kondratyev a développé une méthode de balayage partielle pour les circuits asynchrones de type NCL (Kondratyev, Sorensen, & Streich, 2002).

La plupart des variétés de circuits asynchrones sont, de par leur design, robustes face aux variations de délai. Cela n'affecte bien souvent uniquement que leurs performances et non leurs fonctionnalités. Pour cette raison, peu de recherche de test des pannes de délai ont été effectués par rapport au modèle de faute collé-à sur les circuits asynchrones (Shi & Makris, 2006) .

Après plusieurs approches (Keutzer, Lavagno, & Sangiovanni-Vincentelli, 1991; Lavagno, Lioy, & Kishinevsky, 1994) pour tester toutes les pannes dues au délai avec la technique de balayage complète, Kishinevsky a proposé, lui, d'utiliser une technique de balayage partielle pour les circuits asynchrones de type micropipeline (Kishinevsky, Kondratyev, Lavagno, Saldanha, & Taubin, 1998).

Enfin Gill a réduit la surface de la logique de test en adoptant une stratégie d'insertion de structure de test non intrusive pour les pipelines linéaires et très économiques pour les pipelines avec des embranchements (Gill, Agiwal, Singh, Feng, & Makris, 2006).

1.4.1.3 État des processus de testabilité automatisé pour circuit asynchrone

Bien que les outils et processus de testabilité automatisés pour circuit asynchrone se fassent rares, ils ne sont pas inexistants. Les clients pour le marché des outils de conception asynchrone sont peu nombreux et par conséquent, après quelques recherches, on se rend rapidement compte qu'aucune solution commerciale viable n'existe actuellement. Les entreprises intéressées par le paradigme asynchrone développent très souvent leurs propres outils ou scripts en s'aidant des logiciels de CAO électronique synchrone disponible sur le marché. On pourrait par exemple citer Octasic, qui, pour développer leur DSP asynchrone Octarm, ont utilisé un outil conçu en interne, Octimize, pour réaliser l'insertion et les liaisons

des bascules à balayage ainsi que l'analyse temporelle. À travers quelques articles (F. te Beest & Peeters, 2005; F. T. Beest, Peeters, Verra, Berkel, & Kerkhoff, 2002), le laboratoire de recherche de Philips, laisse quand même entrevoir quelques morceaux de leur solution *Handshake technology* qui a permis de donné naissance au HT-80C51, un microcontrôleur asynchrone de type micropipeline très basse consommation (HandshakeSolutions, 2004). Handshake Solutions a été la première compagnie à offrir une solution commerciale de design complète pour créer et tester des circuits asynchrones. La technologie a depuis été acquise par NXP lors du rachat de Philips Semiconductor, à ce jour aucune nouvelle solution n'a émergé de cette acquisition.

Le processus de testabilité d'un circuit peut être décomposé en deux sous-parties : la modification du circuit, qui est un travail nécessaire pour rendre la logique testable, puis le test en lui-même, c'est-à-dire, la détection des pannes.

La seconde partie est très souvent bien automatisée car réalisée à l'aide de solution commerciale d'ATPG. Par exemple les travaux de Shi (Shi & Makris, 2006) et Kondratyev (Kondratyev et al., 2002) utilisent TetraMAX de Synopsys tandis que Satagopan (V. Satagopan et al., 2007) utilise FastScan, devenue Tessent de Mentor.

Dans la première partie, l'insertion de la chaine de balayage peut être facilement réalisée à l'aide des outils mentionnés précédemment. En revanche, les autres changements à apporter au circuit doivent faire l'objet de modifications manuelles ou de programmation de scripts spécifiques au design. Ainsi, dans les travaux de Satagopan (Venkat Satagopan et al., 2005) sur l'automatisation du test des circuits QDI de type NCL (Null Conventionnal Logic), trois scripts PERL sont utilisés pour insérer diverses structures de test dans les circuits. De même que les ajouts et modifications nécessaires au circuit pour exploiter les méthodes de test de Philips sont directement intégrés dans la suite logicielle de Handshake Solution TiDE (Timeless Design Environment).

Les circuits asynchrones contiennent très souvent des boucles combinatoires nécessaires à leur fonctionnement nominal, mais qui posent des problèmes lors du test. Une des modifications indispensables au circuit est donc de couper ces boucles. Par exemple, on peut insérer un registre à balayage pour rendre la boucle contrôlable par la chaine de balayage (Zeidler & Krstić, 2015). Une deuxième solution consiste à rendre la boucle directement contrôlable par l'ATPG en insérant un ou exclusif relié à un signal d'entrée externe (V. Satagopan et al., 2007).

Le tableau 1 résume les principales méthodes de test de notre revue de littérature. Ce résumé ne contient volontairement pas toutes les méthodes existantes. En effet, beaucoup de variations de ces techniques existent et un choix a dû être fait pour présenter une image concise du test de circuit asynchrone par balayage. L'objectif est ici de statuer sur le niveau d'automatisation de la testabilité actuelle des différentes techniques de test existantes. Sur sept méthodes choisies, on peut d'ores et déjà remarquer que la moitié possède des solutions d'automatisation convaincante. Parmi elles, on trouve les méthodes de Beest et Shi. Ces deux méthodes ont pour particularité d'utiliser le processus de conception de Handshake Solutions, l'une des seules solutions industrielles déjà commercialisées dans le domaine. Les deux techniques travaillent avec le même type de circuits asynchrones, mais sont chacune spécialisée dans un modèle de panne. La méthode de Satagopan basée sur le travail de Kondratyev se concentre quant à elle sur les circuits insensibles au délai de type NCL. La méthode a pour avantage d'être entièrement automatisée et le processus est décrit dans son article (Venkat Satagopan et al., 2005). La méthode de Kishinevsky nous délivre quelques algorithmes relatifs au test sans toutefois rentrer dans les détails de l'automatisation, l'article se focalise plus à expliquer le fonctionnement de la technique de test ainsi que la génération des patrons de test associé. Petlin, lui, reste muet quant à l'automatisation de son procédé. Enfin, la méthode de test de Al-Terkawi Hasib testé lors du développement de l'Octarm, un DSP asynchrone de chez Octasic est, elle, pour l'instant très faiblement automatisée. La méthode se concentre à tester les circuits asynchrones à données groupées sans accusé de réception. L'objectif de ce mémoire sera donc de remédier à cette faiblesse en proposant un

processus de testabilité automatisé. Quelques améliorations de la technique seront aussi développées en l'appliquant à des circuits plus complexes.

Tableau 1.1 Résumé des méthodes de test et leur niveau d'automatisation

Premier auteur	Article / Méthode de test	Type de circuit asynchrone	Langage de conception	Modèle de pannes	Niveau d'automatisation
Kondratyev	Testing of asynchronous designs by "inappropriate" means. Synchronous approach	NCL/ QDI	VHDL	Collé-à	Bas
Petlin	Scan testing of Micropipeline	Micropipelines/ Donnée groupée avec protocole de handshake	Inconnu	Délai et Collé-à	Bas
Beest	A multiplexer based test method for self-timed circuits	Handshake Solution / Donnée groupée avec protocole de handshake	Haste	Collé- à	Élevé
Kishinevsky	Partial-scan delay fault testing of asynchronous circuits	Micropipeline qui utilise un protocole de handshake	Inconnu	Délai	Moyen
Shi	Testing Delay Faults in Asynchronous Handshake Circuits	Handshake Solution / Donnée groupée avec protocole de Handshake	Haste	Délai	Élevé
Satagopan	DFT Techniques and Automation for Asynchronous NULL Conventional Logic Circuits	NCL/ QDI	VHDL	Collé- à	Élevé
Hasib	Exploiting built-in delay lines for applying *launch-on-capture* at-speed testing on self-timed circuits	Octasic/ Données groupées sans protocole de handshake	VHDL	Délai	Bas
Lambert	Ce mémoire	Octasic / Données groupées sans protocole de handshake	VHDL	Délai	Élevé

1.4.2 Circuits asynchrones de type Octasic

1.4.2.1 Fonctionnement de la logique

La logique des DSP asynchrones d'Octasic est particulière puisque son fonctionnement va à contrecourant des méthodes de conception asynchrone les plus répandues. Ici, on ne parlera pas de protocole à 2 ou 4 phases puisque la logique est tout simplement dénuée de méthode de *handshaking.* Comme dans la logique synchrone, le codage du bus de données est sur un fil. La synchronisation des bascules compte sur des lignes à délais pour retarder l'horloge relativement au temps de propagation de la logique combinatoire entre deux éléments séquentiels. Pour que la donnée soit valide en sortie de l'étage, le délai de la ligne à délai CDL doit être supérieur au temps de propagation nécessaire pour traverser la logique combinatoire CLC entre les deux bascules ainsi que certaines marges comme les temps de prépositionnement et de maintien.

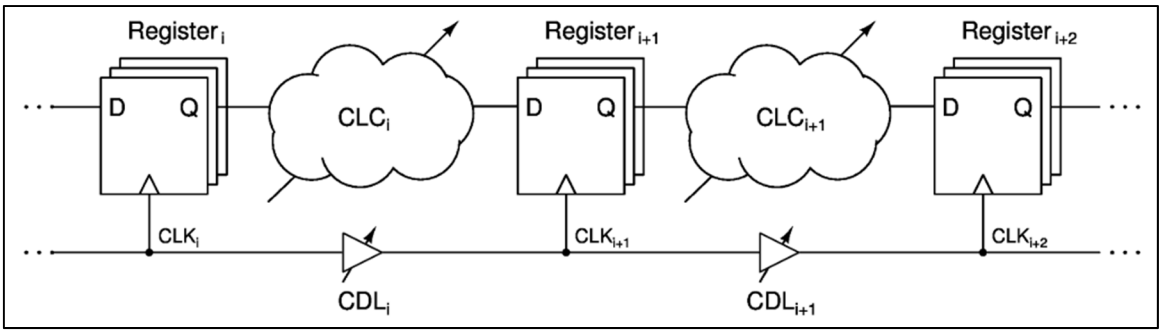

Figure 1.12 Exemple généralisé de la logique asynchrone utilisée par Octasic
Tirée de (Hasib et al., 2018)

Dans son article (Laurence, 2012), Michel Laurence, PDG d'Octasic, a présenté une architecture de processeur qui utilise le même principe de circuit asynchrone que nous venons de présenter. En 2012, cette architecture permettait au produit d'Octasic (OCT2200)

d'être énergiquement 3 fois plus efficace que le TI C6472, le meilleur DSP de Texas Instrument dans sa catégorie.

1.4.2.2 Méthode de test

Une méthode pour tester à vitesse nominale les circuits asynchrones utilisés par Octasic est publiée en 2018 (Hasib et al., 2018). La technique est la première à tester ce type de logique à l'aide de chaînes de balayage pour tester les pannes dues au délai puisque les DSP d'Octasic ont pour l'instant seulement été testés de manière structurelle avec le modèle collé-à. Le principe de la méthode est d'appliquer un test *launch-on-capture* à vitesse nominale en tirant partie des lignes à délais du circuit.

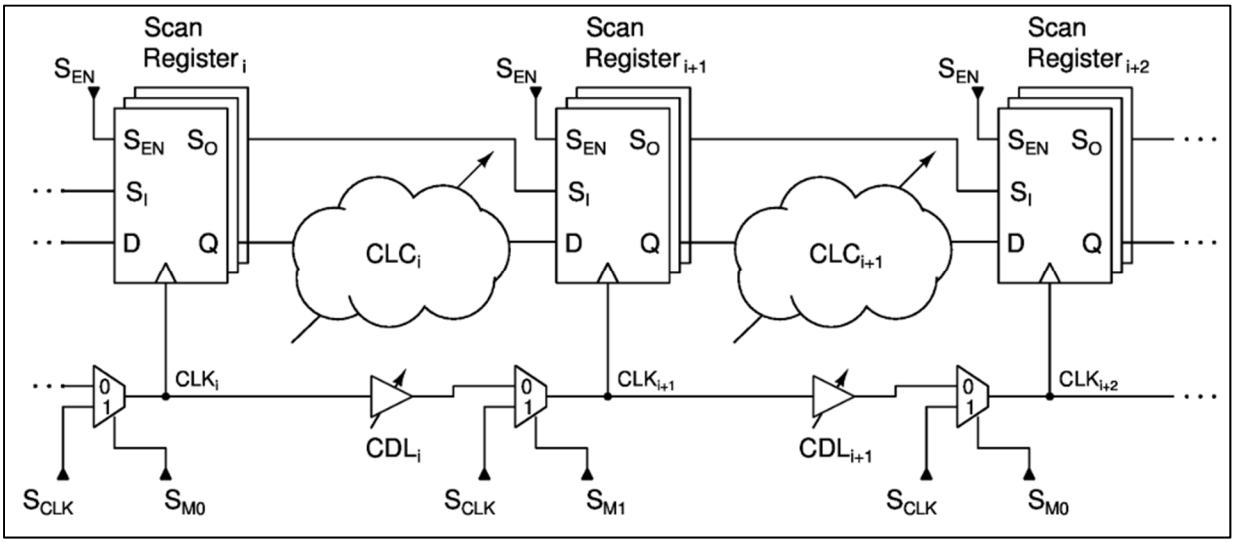

Figure 1.13 Exemple généralisé de circuit asynchrone utilisé par Octasic avec l'intégration des structures de test
Tirée de (Hasib et al., 2018)

Pour appliquer le test, des multiplexeurs sont insérés à chaque domaine d'horloge du circuit. Ceux-ci permettent de choisir quelle horloge déclenche les registres associés. Ainsi, si on active le signal de sélection de mode SM_i à 1, l'horloge provient d'un signal extérieur S_{CLK} ,et quand SM_i est à 0 , CLK_i est générée par l'horloge CLK_{i-1} retardée du temps de propagation

de la ligne à délai. De plus, comme dans la logique synchrone, les registres sont remplacés par leurs équivalents testables afin de former une chaine de balayage.

Ces signaux de contrôle et structure supplémentaire permettent d'appliquer la stratégie de *launch-on-capture* présentée sur la Figure 1.14 .

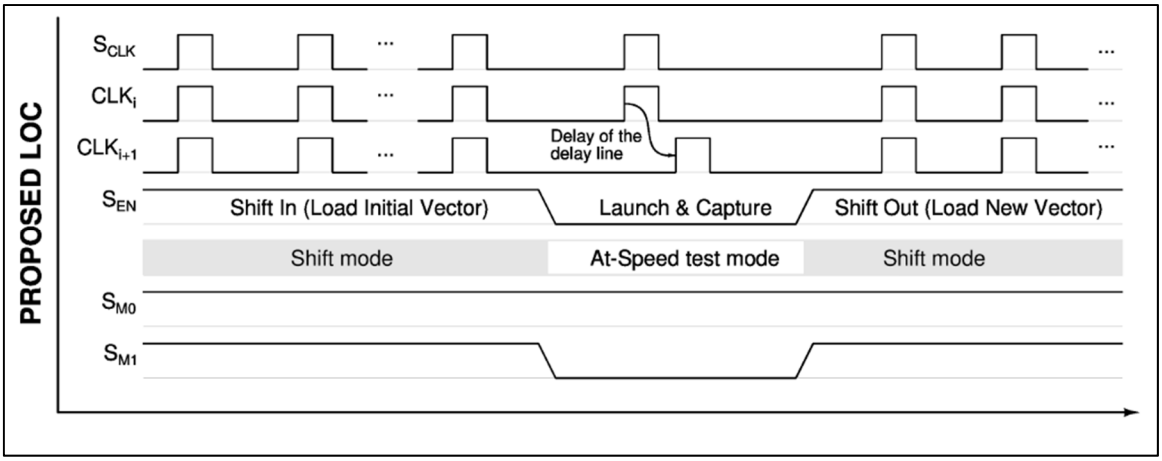

Figure 1.14 Chronogrammes de la stratégie de *launch-on-capture* a vitesse nominale
Tirée de (Hasib et al., 2018)

Comme n'importe quelle technique qui utilise les bascules à balayage, on commence par charger le vecteur dans la chaîne. Pour ce faire, le signal S_{EN} est mis à 1 et tous les multiplexeurs sont réglés pour choisir l'horloge S_{CLK}. Chaque impulsion sur le signal S_{CLK} charge et décale le vecteur dans la chaine de balayage.

Pour lancer le *launch-on-capture*, on règle les multiplexeurs de manière à ce qu'un domaine d'horloge sur deux reçoive le signal S_{CLK} tandis que l'autre moitié acquiert l'horloge retardée par la ligne à délai comme sur la Figure 1.15.

Pour couvrir toute la logique, la procédure doit être répétée une deuxième fois en inversant les rôles des registres. Ainsi les bascules auparavant utilisées pour le lancement sont utilisées pour la capture et vice versa.

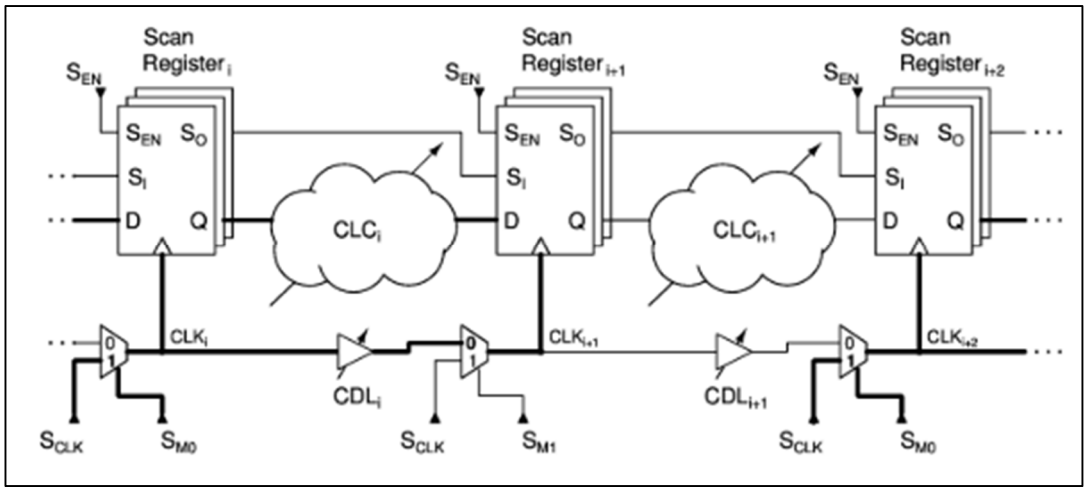

Figure 1.15 Exemple généralisé de circuit asynchrone utilisé par Octasic avec l'intégration des structures de test, les signaux mis en évidence sont ceux utilisés lors de l'application du *launch-on-capture*
Tirée de (Hasib et al., 2018)

1.4.2.3 QMI : outil de découverte des points de convergence

L'automatisation de la méthode de test présentée demande d'établir des groupes de registres qui peuvent être contrôlés par le même signal de sélection de mode SM_i. Une des étapes importantes pour réaliser ces regroupements est la recherche de point de convergence entre les chemins d'horloge et le chemin de donnée. On définit un point de convergence comme un endroit du circuit où, à partir d'un domaine horloge, les chemins d'horloge et de donnée convergent sur un même registre (Figure 1.16). Cette problématique a déjà fait l'objet de précédents travaux et a mené à la conception d'un outil de découverte des points de convergence, QMI (TÊTU, 2014). En parcourant le circuit représenté sous la forme d'un graphe, le programme est capable de lister tous les points de convergence d'un circuit. Cependant, même si l'outil fonctionne parfaitement, son temps d'exécution est inadapté pour un contexte d'utilisation réelle. En effet, il lui faut compter plus de 60 jours pour analyser un circuit contenant 3 domaines d'horloge et une quarantaine de registres (TÊTU, 2014). Ces tests sont réalisés sur une plateforme équipée d'un processeur Xeon E78870 d'Intel cadencé à 2,40 GHz équipé de 1 To de mémoire vive.

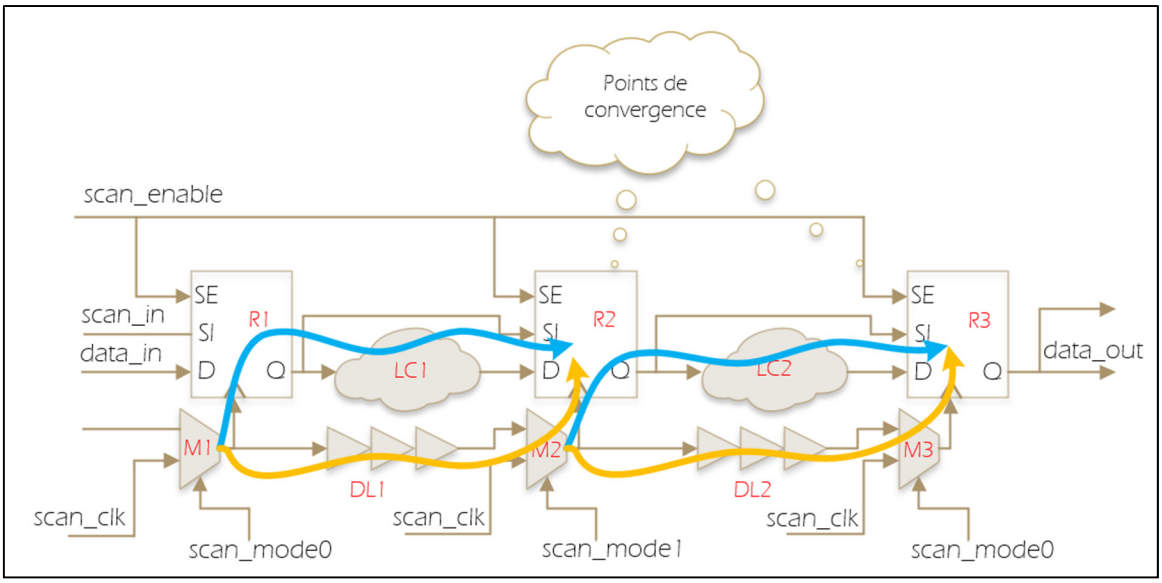

Figure 1.16 Représentation des points de convergences, les chemins de données (en bleu) et les chemins d'horloges (en orange) se rassemblent aux points de convergences R2 et R3

1.5 Conclusion

À travers cette revue de littérature, nous avons vu que la testabilité des circuits asynchrones pose des défis majeurs. La diversité des types de logique asynchrone a inexorablement entrainé une grande variété de méthodes de test et d'autant plus de manières de les automatiser. Malgré tout, on a aussi pu constater que les auteurs se consacrent davantage sur les techniques que sur l'automatisation des méthodes de test. Le sujet de l'automatisation est pourtant un frein majeur à l'adoption de la technologie asynchrone. L'explosion des domaines comme l'intelligence artificielle qui demande beaucoup de puissance de calcul et la concurrence impose aux concepteurs de circuits intégrés des cadences de développement importantes. On leur demande de faire toujours plus, et ce, avec le même temps de développement qu'il leur avait été alloué lors de leur précédentes innovations. Nous devons donc faire en sorte que l'ensemble des activités de conception, de la première ébauche du design au test du circuit, soit un processus rapide et fortement automatisé. Pour autant, nous avons aussi vu qu'une partie de ces méthodes de test ont un niveau élevé d'automatisation. C'est notamment le cas des techniques développé par HandShake Solutions qui sont toutes

intégrées dans leur suite d'outils de développement à visée commercial. Cependant toutes ces solutions concernent des architectures de circuits asynchrones différentes de celle étudiée. L'architecture d'Octasic et la méthode de test sur lesquels nous travaillons sont relativement nouvelles. Les premiers produits commerciaux à avoir utilisé ce type de logique sont l'OCT1010 et l'OCT2200, des DSP de Octasic sorties successivement en 2008 puis 2011. La méthode de test pour la couverture des pannes de délai est seulement publiée en 2018, soit 10 ans après la commercialisation du premier circuit. Le travail de ce mémoire est donc la première ébauche de l'automatisation de cette nouvelle méthode.

CHAPITRE 2

BANC D'ESSAI

2.1 Introduction

L'objectif de ce chapitre est de présenter tous les circuits qui nous ont aidés à améliorer et automatiser le processus de testabilité des circuits de type Octasic. Nous commencerons par décrire le fonctionnement du générateur d'horloge pour circuit asynchrone qui nous a été utile pour fournir une horloge à nos circuits les plus simples. Puis, nous présenterons trois types de circuits : les machines à états finis issues du jeu de circuits ITC99, un pipeline asynchrone typique de l'architecture d'Octasic et un mini-MIPS asynchrone dérivé de l'architecture d'Octasic. Les circuits issus du benchmark ITC99 nous ont aidés à améliorer le processus de testabilité. En effet, c'est en se basant sur ces circuits que nous avons pu proposer une méthode pour tester les machines à états finis asynchrones qui sera décrite dans un autre chapitre. Le pipeline asynchrone typique d'Octasic est le circuit qui nous a permis de développer la base de l'automatisation du processus de test. Enfin le mini-MIPS asynchrone est le circuit le plus difficile sur lequel nous avons travaillé. Son architecture complexe nous a aidés à améliorer, optimiser et ajouter des fonctionnalités au processus existant.

2.2 Générateur d'horloge pour circuit asynchrone

Le circuit de génération d'horloge est utilisé pour délivrer un signal d'horloge dans nos bancs d'essai les plus simples, c'est-à-dire le pipeline asynchrone et les quelques circuits de machine à états finis choisis du benchmark ITC99. Il est composé de deux parties : l'unité de contrôle du jeton et le générateur d'impulsion.

Lorsque le signal de réinitialisation est relâché, le niveau logique présent à la 2èmeentrée du *non-ou*, qu'on appelle « jeton », est inversé et se propage jusqu'à l'entrée *D* du registre

(Figure 2.1). Dans un même temps, une impulsion est créée à l'aide d'un *ou-exclusif* en comparant le jeton et le jeton retardé par *GIDelay* à l'aide d'une première ligne à délai. Cette impulsion se propage ensuite dans une deuxième ligne à délai *ExtDelay* pour venir déclencher le registre et donc inverser le jeton pour démarrer un nouveau cycle.

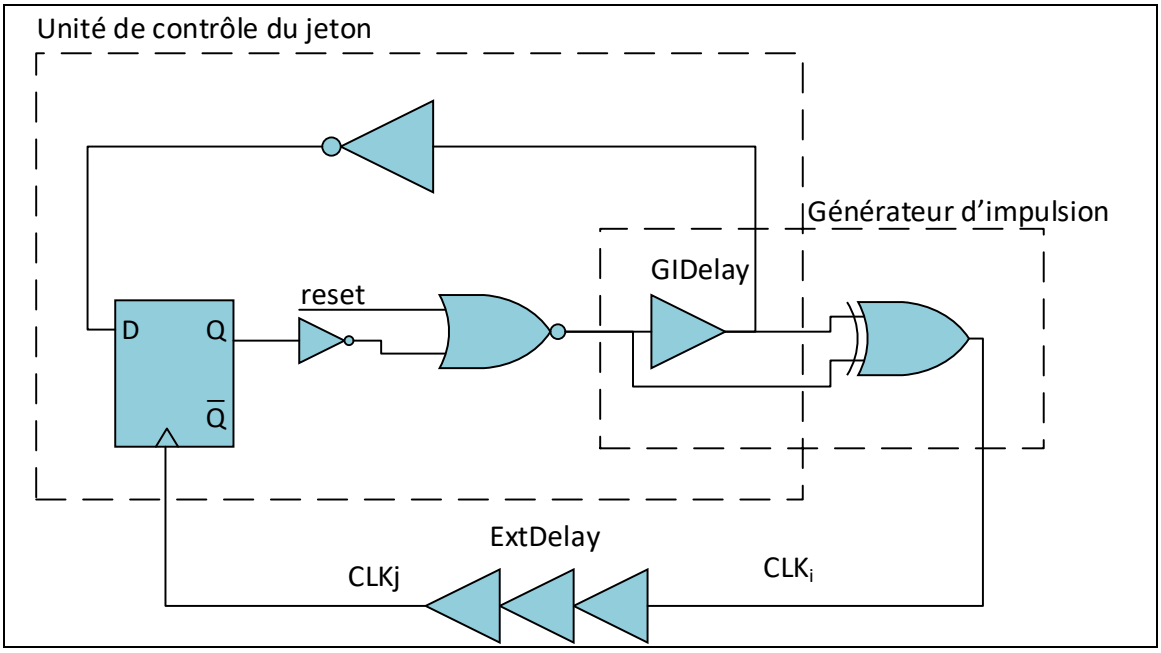

Figure 2.1 Générateur d'horloges pour circuit asynchrone

La largeur de l'impulsion est déterminée par la taille de GIdelay de telle sorte que :

$$T_{impulse} = T_p(GIDelay), \text{ où } Tp \text{ correspond au temps de propagation} \quad (2.1)$$

La période d'horloge correspond essentiellement aux délais de propagation dans les deux lignes à délai.

$$T_{horloge} = T_P(GIDelay) + T_p(ExtDelay) + \alpha \quad (2.2)$$

Le terme α correspond à tous les autres délais du chemin.

On comprend alors, que si l'on souhaite que l'horloge ait un rapport cyclique de moins de 50%, il est important que $T_p(ExtDelay)$ soit deux fois plus important que $T_p(GIDelay)$.

2.3 Circuit de test

2.3.1 Machines à états finis issues de ITC99

Pour évaluer l'efficacité de notre processus de test de machines à états finis asynchrones, trois circuits ont été choisis dans le jeu de test ITC99. Le benchmark ITC99 regroupe un ensemble de circuits au niveau RTL dont les caractéristiques sont typiques des circuits synthétisés. Il a été développé par le groupe de recherche « Electronic CAD & Reliability Group » à l'école italienne « Politecnico di Torino ». Son but est d'offrir des circuits de référence aux développeurs d'algorithme d'ATPG (Corno, Reorda, & Squillero, 2000). L'avantage d'un tel choix est qu'il nous permet de comparer nos résultats de taux de couverture de panne avec de précédents travaux réalisés sur ce benchmark (Tableau 1) (Touati, 2016).

Tableau 2.1 Résultat de taux de couverture de panne sur les modèles de test collé-à et *Launch-on-capture* sur les circuits b01, b02 et b06 du jeu de test ITC99
Tiré de Touati (2016)

Circuit	**#Registres**	**Collé-à**		***Launch-on-capture***		**Fonction du circuit**
		#patrons	TC (%)	#patrons	TC (%)	
b01	5	23	100	18	69.91	Comparateur de ligne série
b02	4	17	100	13	82	Détecteur de nombre BCD
b06	8	17	100	13	69.82	Gestionnaire d'interruption

Le banc d'essai sera composé du générateur d'horloge précédemment présenté et d'une des machines à états finis du Tableau 2.1. Les modules seront connectés comme décrit à la Figure 2.2 pour former une machine à états finis asynchrone.

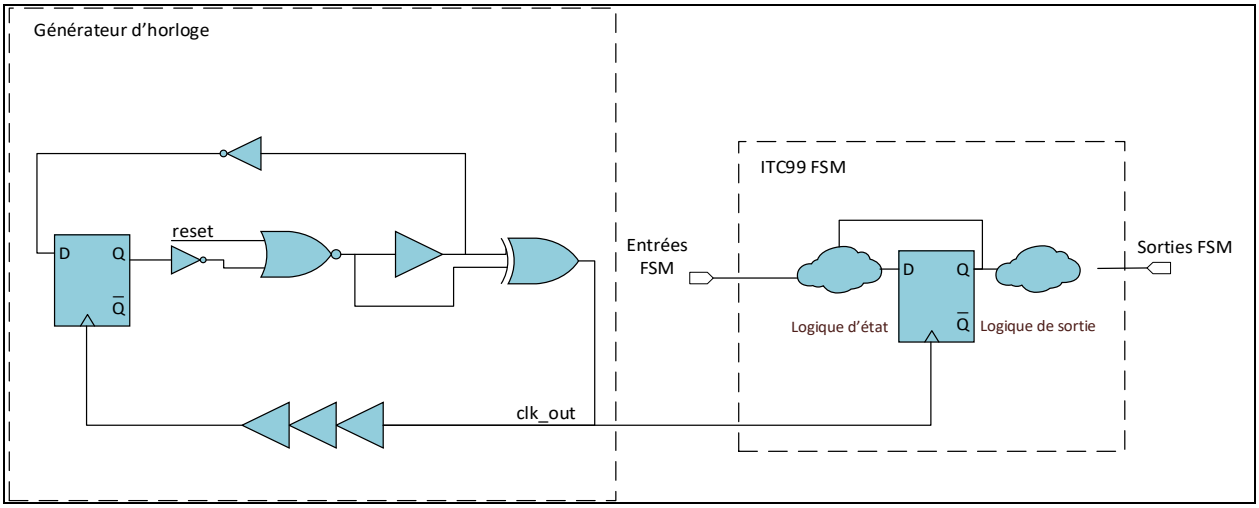

Figure 2.2 Banc d'essai composé du générateur d'horloge et d'une machine à états finis issue du jeu de circuit ITC99

2.3.2 Pipeline typique des circuits d'Octasic

Conçu pour être au plus proche de l'architecture typique de Octasic, le simple pipeline asynchrone (Figure 2.3) nous permet de développer et tester les bases du processus de test automatisé. Il est constitué de 3 étages intégrant de simples multiplicateurs en guise de logique combinatoire. L'horloge provenant du générateur est retardée 3 fois afin de successivement déclencher les différents étages. La taille des lignes à délai a été calculée en fonction du délai dans le chemin de données correspondant.

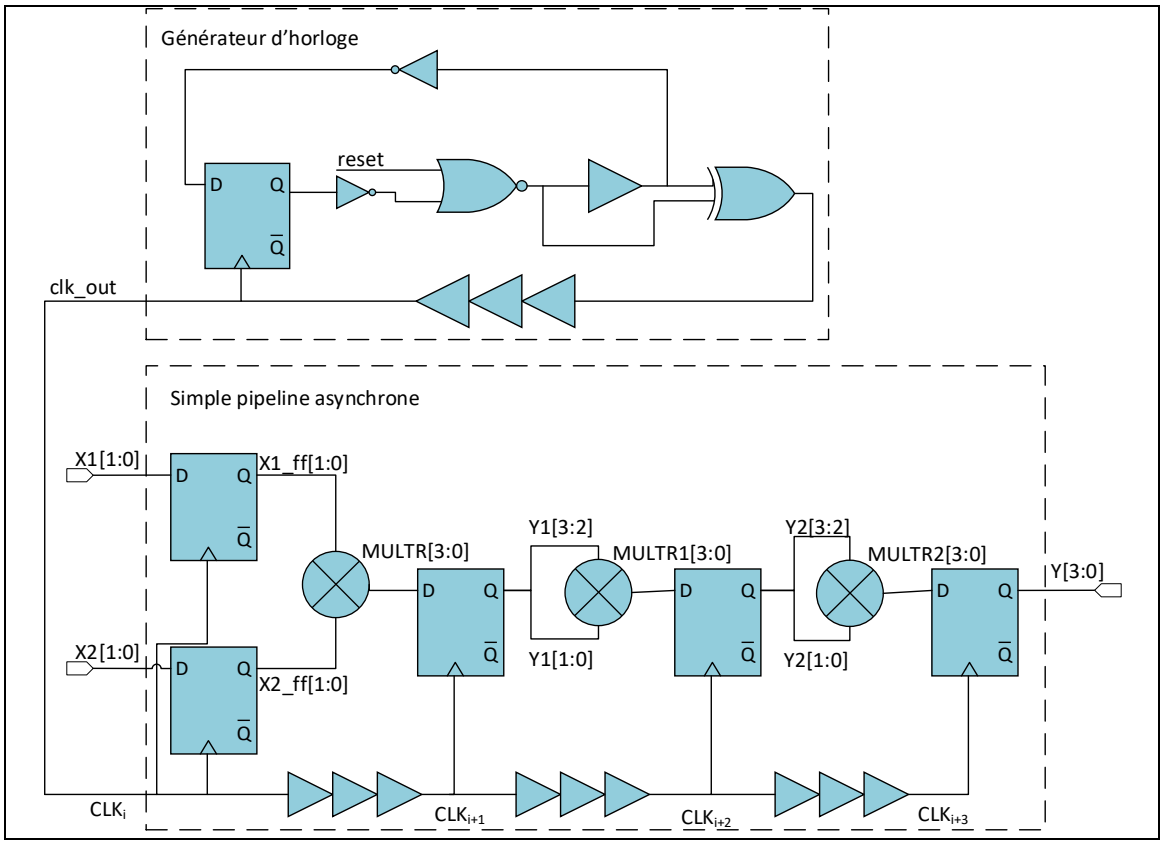

Figure 2.3 Le simple pipeline asynchrone connecté au générateur d'horloge

2.3.3 Microprocesseur asynchrone mini-MIPS

Avec près de 1000 registres et 18 domaines d'horloge, le mini-MIPS est le circuit le plus complexe sur lequel nous avons travaillé. Le circuit à l'origine conçu pour FPGA a été développé dans l'objectif de formaliser la technique de conception d'Octasic lors de précédents travaux (Mickaël Fiorentino, 2017b). C'est en réalité plutôt un dérivé de la technique de conception d'Octasic. En effet, le circuit utilise un agencement du réseau d'horloge qui a permis de faciliter l'automatisation de sa conception, mais en contrepartie complique grandement la possibilité de tester le circuit.

2.3.3.1 Architecture

Le processeur est constitué de 5 ressources (Figure 2.4): le compteur du programme (*PC*), la mémoire d'instruction (*Imem*), la banque de registre (*RF*), l'unité de calcul *(ALU*) et la mémoire vive (*Dmem*). Ces 5 ressources sont partagées par l'intermédiaire d'un commutateur (*Crossbar-Switch*) aux 3 unités d'exécution (*Execution Unit - EU*).

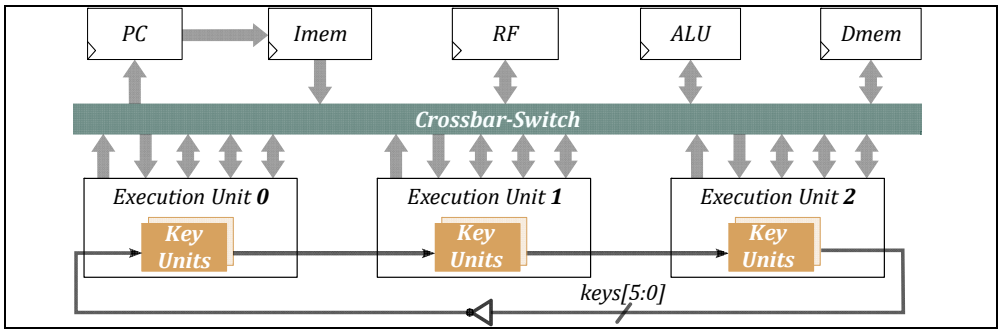

Figure 2.4 Vue d'ensemble du mini-MIPS
Reproduite et adaptée avec l'autorisation de (Fiorentino, 2018)

Les EU communiquent ensemble à l'aide de ce qu'on appelle, le *keyring*, qui fait référence aux dépendances d'horloges en anneaux. Celui-ci est composé de 6 bits représentant chacun une ressource à l'exception de la banque de registre qui accapare 2 bits. Ces 6 bits, qu'on appelle *keys*, permettent aux EU d'exécuter les instructions en même temps tout en partageant les mêmes ressources. Chaque unité d'exécution dispose donc de toute la logique qui permet d'interpréter toutes les instructions du programme.

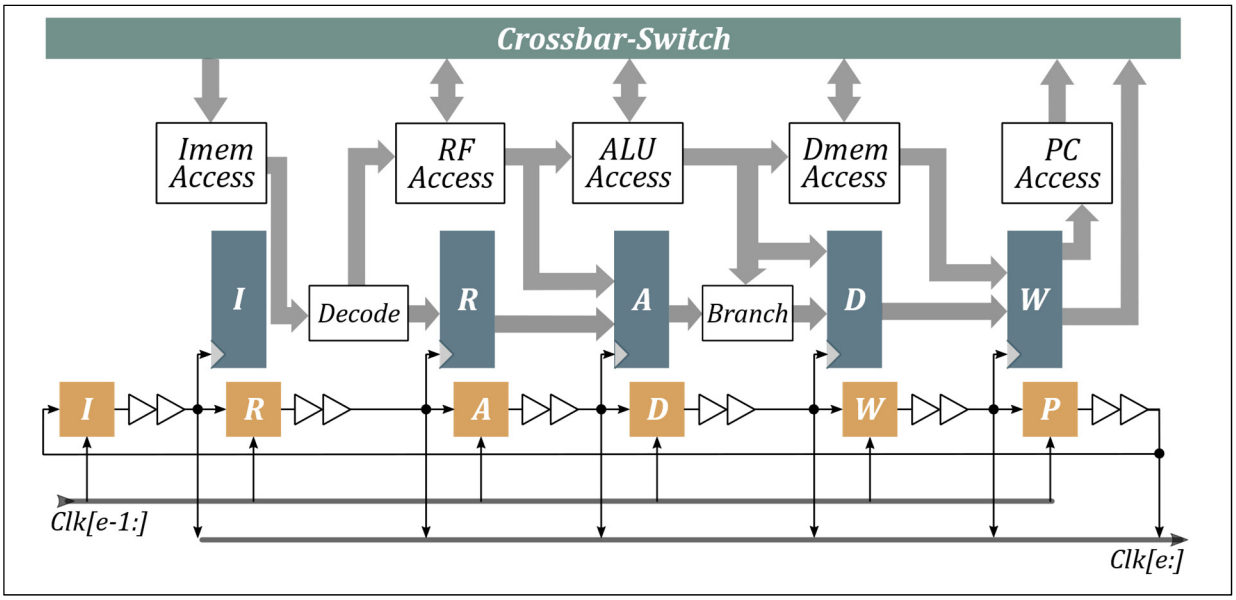

Figure 2.5 Schéma représentatif d'une unité d'exécution, les key units sont représentées en jaune
Reproduite et adaptée avec l'autorisation de (Fiorentino, 2018)

Comme le montre la figure 2.5, une unité d'exécution contient 6 *key units* représentées en jaune. Celles-ci sont directement connectées à des lignes à délai dont leurs tailles sont calibrées sur le délai des chemins de données correspondant. Cependant, dans cette version du mini-MIPS, une seule taille de ligne à délai a été choisie en fonction du délai maximum des chemins de données.

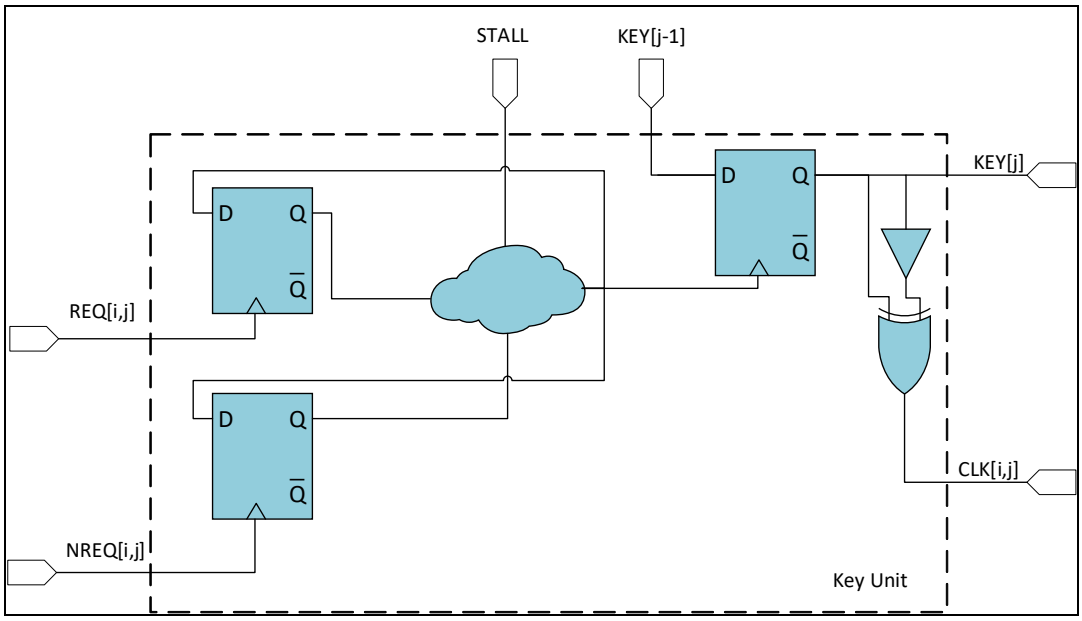

Figure 2.6 Schéma d'un key unit

Le rôle du *key unit* (Figure 2.6) est de générer une horloge locale lorsque la ressource correspondante est disponible. Pour qu'une impulsion soit générée sur la sortie *CLK[i,j]*, il faut que la clé soit arrivée à l'entrée *KEY[j]* et que les horloges dont dépend le *key unit* soient déclenchées sur les entrées *REQ[i,j]* et *NREQ[i,j]*. L'entrée *STALL* est activée lorsque le système a besoin de mettre en pause le *key unit* , par exemple, lors d'un calcul un peu long dans l'ALU. La dépendance entre les *key units* est représentée sur la Figure 2.7.

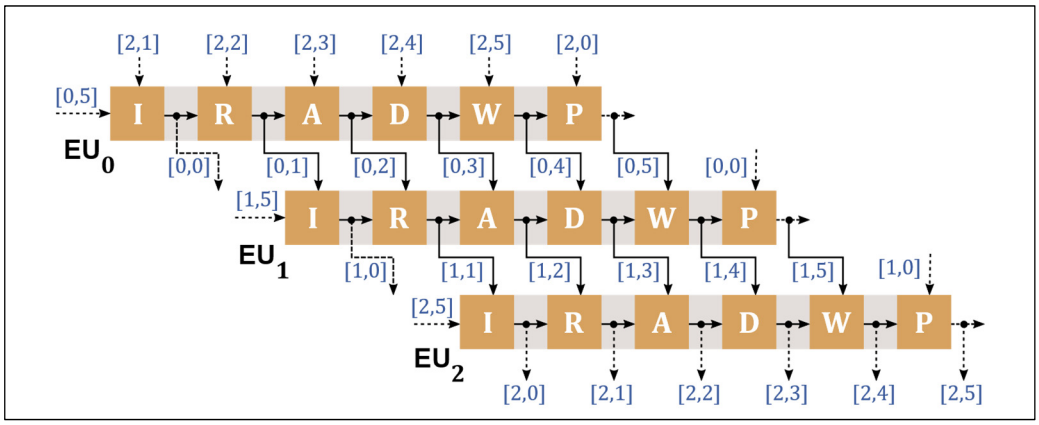

Figure 2.7 Schéma représentant la dépendance entre les key units
Reproduite et adaptée avec l'autorisation de (Fiorentino, 2018)

2.3.3.2 Détails d'implémentation

Lors de la conversion du design pour FPGA vers la librairie de la technologie ASIC 45nm de Cadence, nous avons choisi d'extraire tous les modules contenant des banques de mémoire tels que la mémoire d'instruction, la banque de registre et la mémoire vive. Tous ces modules ont été déplacés dans le banc de test (Figure 2.9). Ce choix a été fait, car le test de banque de mémoire ne peut pas être réalisé avec les mêmes techniques que le reste du circuit. En effet, pour des raisons de consommation et d'ajout de délai additionnels, les registres des banques de mémoire ne peuvent pas être remplacés simplement par des registres à balayage (Belete et al., 2002; Zeng & Abadir, 2004). Ce choix a pour avantage de prévenir l'impact négatif des banques de mémoire sur le taux de couverture de pannes du circuit. Le désavantage est que la suppression des modules mémoire du circuit a eu pour effet de supprimer les registres de capture du commutateur du mini-MIPS. On se retrouve donc avec une partie de la logique combinatoire directement connectée sur des sorties externes du processeur (Figure 2.8). Cette logique ne pourra pas être testée en *launch-on-capture* et affectera donc aussi notre taux de couverture de pannes global.

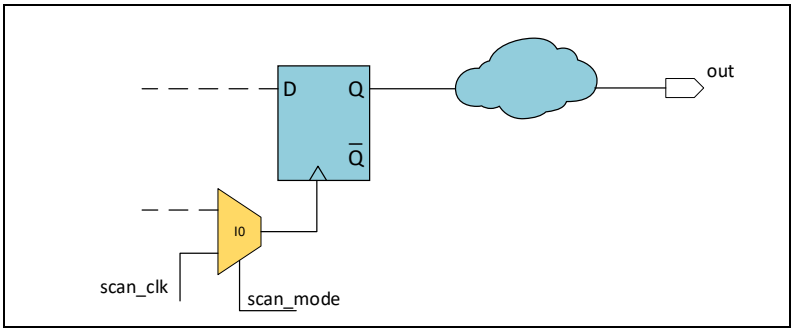

Figure 2.8 Logique combinatoire sans registre de capture, la logique combinatoire est directement reliée à la sortie sans registre de capture, ce qui rend ses pannes indétectables en *launch-on-capture*

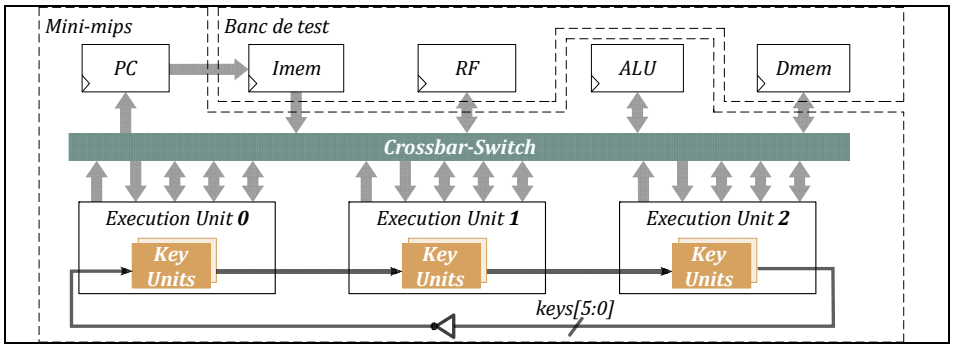

Figure 2.9 Vue d'ensemble du mini-MIPS et son banc de test
Reproduite et adaptée avec l'autorisation de (Fiorentino, 2018)

2.4 Conclusion

Trois types de circuits ont été présentés dans ce chapitre : les machines à états finis du jeu de circuits de référence ITC99, le simple pipeline typique de l'architecture d'Octasic et le mini-MIPS asynchrone. L'utilisation de ces circuits a deux types d'objectifs. Le premier est de permettre l'amélioration du processus de test par l'ajout de support de nouvelles configurations de circuit. C'est ce que permet le jeu de circuit ITC99 en nous aidant à développer le support de test de machine à états finis asynchrone. Le deuxième objectif est d'aider au développement de l'automatisation du processus de testabilité. C'est le pipeline d'Octasic et le mini-MIPS qui nous aident à atteindre ce but. Le tableau 2 résume les caractéristiques des circuits que nous avons présentées. Ces résultats sont extraits de la synthèse des circuits à l'aide de la technologie éducative 45nm et le logiciel Genus de Cadence. Ils nous permettent de comparer la complexité des circuits et nous donner une référence qui sera utile pour mesurer l'impact de l'insertion des structures de test sur la surface du circuit. Avec 988 registres, 4570 portes et 18 domaines d'horloge, le mini-MIPS est sans aucun doute le circuit le plus complexe sur lequel nous avons travaillé. C'est en majorité sur ce microprocesseur asynchrone que s'est déroulé le développement de l'outil d'automatisation du processus de testabilité.

Tableau 2.2 Caractéristiques des circuits de test

Circuits		**Nombre de portes**	**Nombre de registres**	**Surface (μm^2)**	**Nombre de domaines d'horloges**
ITC99	b01	26	5	63	1
	b02	21	4	47	1
	b06	28	8	83	1
Pipeline Octasic		110	17	824	3
Mini-MIPS		4570	988	21 175	18

CHAPITRE 3

STRATÉGIE DE TEST

3.1 Introduction

Le but de ce chapitre est de présenter la stratégie de test pour chaque circuit du banc d'essai précédemment introduit. Dans une première partie, on détaillera l'ensemble des structures de test spécifiques (qui s'ajoutent aux registres à balayage) pour rendre les tests de transition possibles. Ces structures seront présentées dans l'ordre de leur utilité. Ainsi, le multiplexeur de test est utilisé dans tous les circuits du banc d'essai tandis que le module de contournement de *key unit* est uniquement utile au mini-MIPS. La deuxième partie sera chargée de montrer l'intégration de ces structures de test dans les différents circuits. Elle nous apprendra aussi comment l'impulsion de capture est générée par rapport à celle de lancement lors d'un de test de transition. La section sur le mini-MIPS sera décomposée en sous-sections. On apprendra dans la première sous-section que le mini-MIPS étudié comporte 8 sortes de chemins de données et s'écarte donc du modèle d'architecture idéal présenté dans la revue de littérature. La deuxième sous-section permettra de déterminer la stratégie de test mise en place pour tester chaque type de chemin. Enfin, la dernière sous-section ouvrira une discussion sur l'importance du test à vitesse nominale en fonction du type de chemin de données.

3.2 Structure de test spécifique

L'objectif de cette partie est de présenter les trois structures de test additionnelles intégrées dans les circuits du banc d'essai : le multiplexeur de test, le module de test de machine à états finis et la structure de contournement de *key unit.*

3.2.1 Multiplexeur de test

Conformément à la méthode de test (Hasib et al., 2018) décrite en revue de littérature, le multiplexeur de test est placé en aval des lignes à délai (Figure 3.1). Il fait partie des structures de test générique, c'est-à-dire les structures valables pour tous les circuits de type Octasic. Son objectif est double : le premier est de permettre le chargement de la chaîne de balayage à travers le circuit via le signal d'horloge *scan_clk* quand tous les signaux *scan_mode* sont mis à 1. Le deuxième est de pouvoir sélectionner le groupe de registres qui va être déclenché dans l'étape de lancement de la technique de *launch-on-capture*. Ainsi, si l'on souhaite réaliser un test de *launch-on-capture* dans l'étage représenté en Figure 3.1, on règle *scan_mode0* à 1 pour déclencher le lancement par l'intermédiaire de *scan_clk* et on met scan_mode1 à 0 pour qu'il récupère l'impulsion provenant de la ligne à délai pour réaliser la capture.

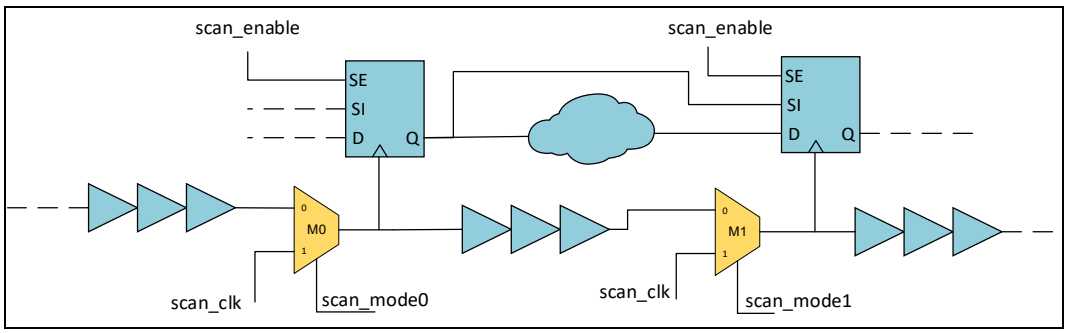

Figure 3.1 Exemple d'utilisation du multiplexeur de test

3.2.2 Module de test de machine à états finis

La création du module *FSM_test* a été nécessaire pour rendre la technique de *launch-on-capture* possible sur les machines à états finis. Ce module fait partie des structures de test qu'on qualifie de spécialisées. Son objectif est de générer une impulsion de capture à partir de l'horloge de lancement en utilisant la boucle d'horloge d'origine du circuit (Figure 3.2).

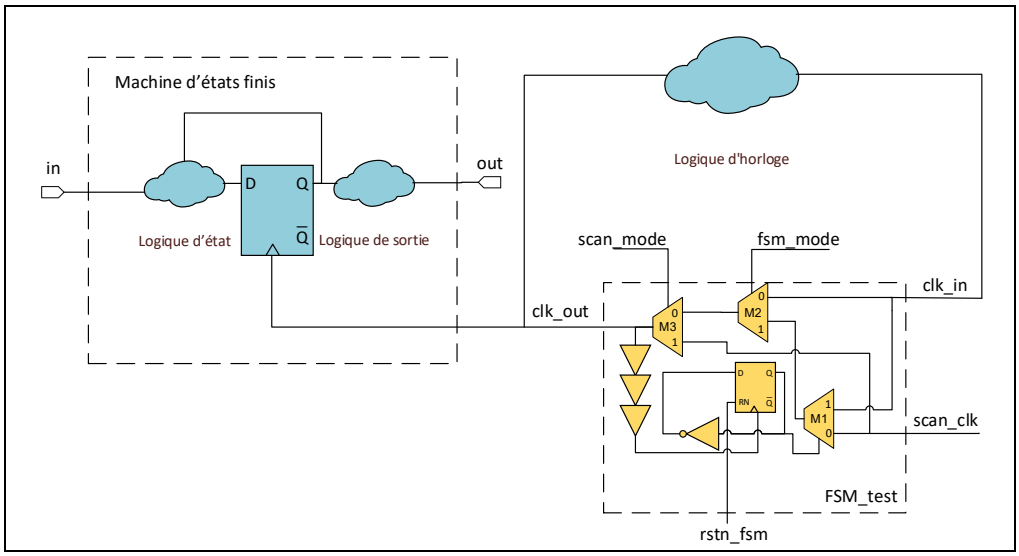

Figure 3.2 Exemple d'utilisation du module *FSM_test*

Le module de *FSM_test* comporte 3 modes de fonctionnement qui peuvent être sélectionnés par l'intermédiaire de deux signaux de contrôle*, scan_mode* et *fsm_mode* :

Le mode normal n'affecte pas la logique; il permet un fonctionnement nominal du circuit. Le signal *clk_in* est directement reconduit à la sortie *clk_out* en traversant les multiplexeurs *M2* et *M3*.

Le mode de test de FSM est celui qui permet d'exécuter la technique du *launch-on-capture* sur une machine à états finis :

- on initialise d'abord le registre interne au module de test avec le signal *rstn_fsm* pour que l'entrée *scan_clk* soit sélectionnée par *M1*;
- on lance une impulsion sur l'entrée *scan_clk* de *M1*; celle-ci est directement dirigée vers la sortie *clk_out* en traversant *M2* et *M3*; c'est l'impulsion de lancement;
- l'impulsion de lancement est retardée à l'aide d'une ligne à délai et vient inverser le bit de sélection de *M1* pour sélectionner l'entrée *clk_in* en déclenchant le registre interne. La ligne à délai est nécessaire pour que l'impulsion de lancement ne soit pas coupée trop rapidement et que celle-ci est une largeur convenable;

- dans le même temps, l'impulsion de lancement traverse la logique d'horloge du circuit et revient à l'entrée *clk_in*;
- l'impulsion parcourt le module jusqu'à la sortie *clk_out* en traversant *M1*, *M2* et *M3*; c'est l'impulsion de capture;
- l'impulsion de capture revient encore une fois sur l'entrée *clk_in* mais se retrouve cette fois-ci bloquée, car le bit de sélection de l'entrée de *M1* a été renversé.

Le mode de test standard permet au module de se comporter de la même manière que le multiplexeur de test précédemment présenté. L'horloge *scan_clk* traverse directement le multiplexeur *M3* pour se rendre à la sortie *clk_out*. Ce mode de fonctionnement est nécessaire pour charger les vecteurs de test dans la chaine de balayage ou pour déclencher uniquement une impulsion de lancement dans le cas où le registre contrôlé fait aussi partie d'un micropipeline classique.

Le Tableau 3.1 résume les trois modes de fonctionnement.

Tableau 3.1 Résumé des modes de fonctionnement du module de test de machine à états finis *FSM_test*

Mode d'opération	fsm_mode	scan_mode	Commentaire
Comportement normal	0	0	--
Mode de test de FSM	1	0	Exécute une impulsion de lancement et de capture lorsque *scan_clk* est déclenché
Mode de test standard	0	1	Exécute seulement une impulsion de lancement lorsque *scan_clk* est déclenché

3.2.3 Module de contournement de Key Unit

Le module de test *Bypass_KU* (Figure 3.3) a été créé dans l'objectif de contourner les structures logiques complexes dans les chemins d'horloge du mini-MIPS. Il est placé en aval de la structure à contourner et est composé d'un multiplexeur qui permet de sélectionner un signal parmi les 3 sources d'horloge suivantes :

clk_in_bus[10 :0] correspond aux horloges dont la structure contournée est dépendante pour la génération de son impulsion. Le choix de ces horloges est déterminé à l'aide d'un algorithme qui sera détaillé dans le chapitre consacré à l'automatisation du processus de testabilité. C'est cette entrée qui est sélectionnée durant le test du circuit.

default_clk_in est l'entrée correspondante au signal d'horloge provenant de la structure à contourner, c'est cette source qui est utilisée lors du fonctionnement nominal du circuit.

scan_clk est l'entrée correspondante au signal d'horloge nécessaire pour le chargement des vecteurs de test dans les registres dont l'horloge est reliée à la sortie du *Bypass_KU*.

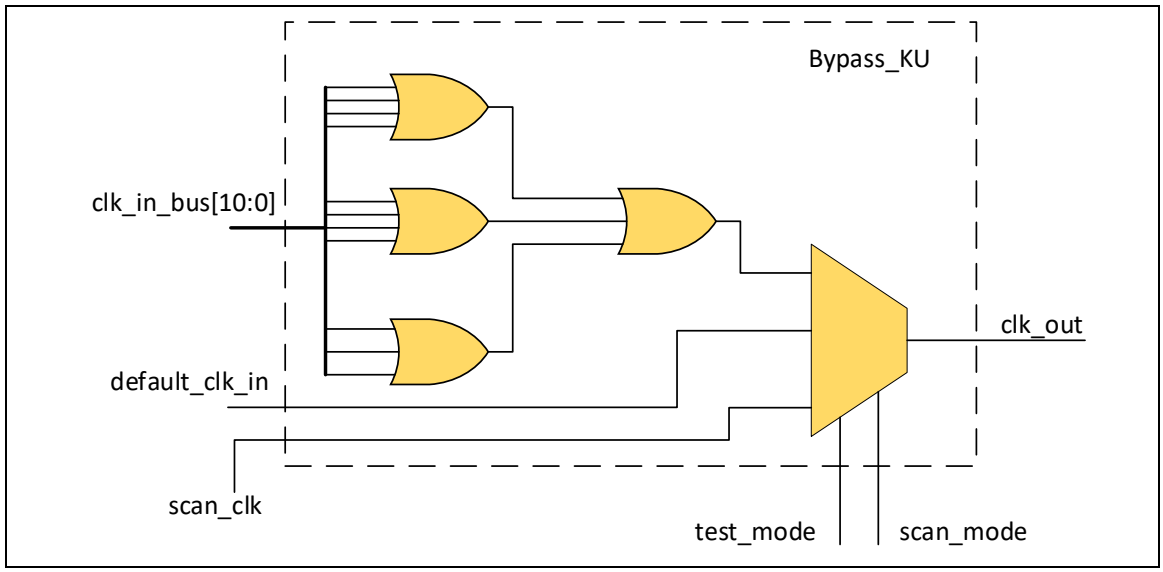

Figure 3.3 Module de contournement *Bypass_KU*

Le Tableau 3.2 résume les modes de fonctionnement du module de contournement *Bypass_KU*.

Tableau 3.2 Résumé des modes d'opération du module de contournement *Bypass_KU*

Mode d'opération	test_mode	scan_mode	Commentaire
Comportement normal	0	0	Sélectionne la source *default_clk_in* dans le cas du fonctionnement nominal du circuit
Mode de contournement des KU	1	0	Sélectionne la source *clk_in_bus[10 :0]*
Mode de chargement	0	1	Sélectionne la source *scan_clk* pour le chargement des vecteurs de test

3.3 Intégration des structures de test spécifique dans le banc d'essai

Le rôle de cette section est de montrer comment les structures de test spécifique précédemment introduites sont intégrées dans les différents circuits du banc d'essai et comment ces intégrations rendent possibles les tests de transition. Les circuits sont présentés dans l'ordre de complexité: le pipeline typique de l'architecture d'Octasic, les machines à états finis du jeu de circuit ITC99 et le mini-MIPS asynchrone.

3.3.1 Pipeline typique de l'architecture d'Octasic

La technique des multiplexeurs de test présenté à la section 3.2.1 est testée sur notre simple pipeline typique de l'architecture d'Octasic. Le processus de testabilité qu'on présentera au CHAPITRE 4 procède au remplacement des bascules par des registres à balayage, l'insertion

et la connexion des multiplexeurs de test. L'insertion des structures de test est illustrée sur la Figure 3.4. On y retrouve quatre multiplexeurs de test (M0 à M3, colorés en vert) et cinq groupes de registres à balayage (colorés en orange).

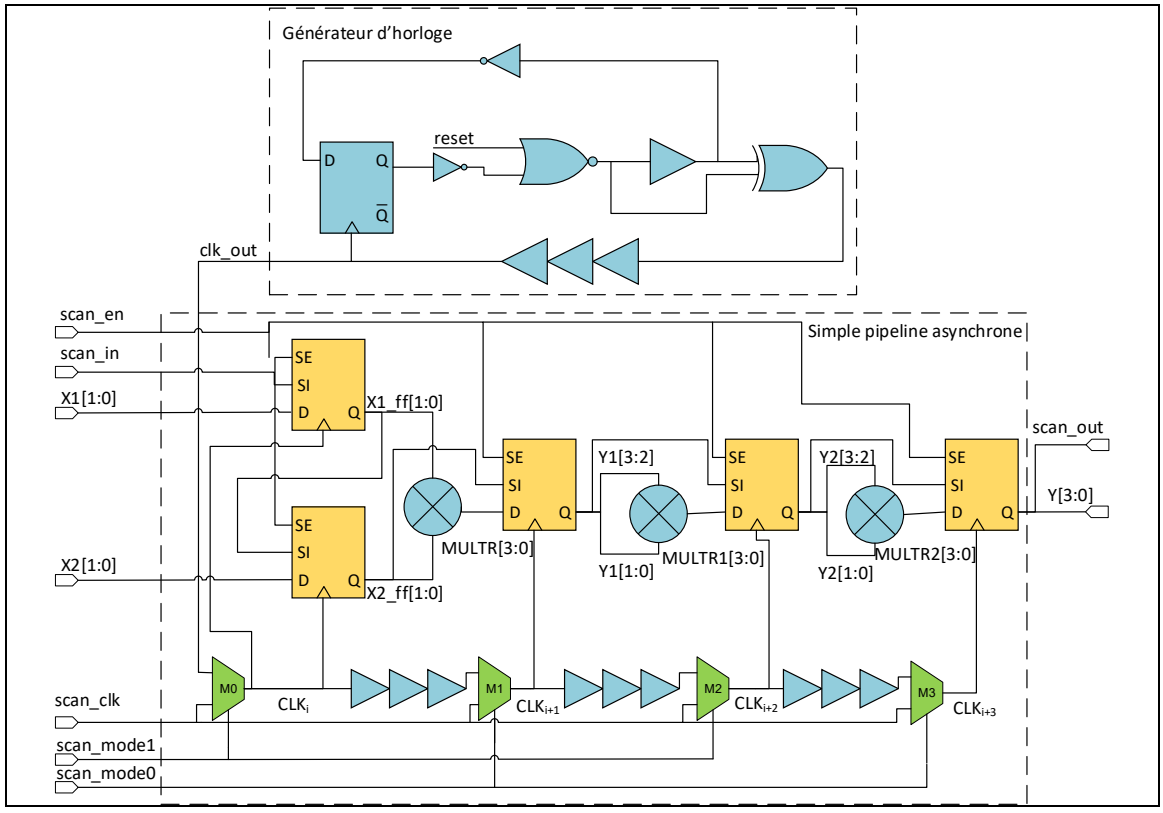

Figure 3.4 Schéma représentant l'intégration des structures de test dans le pipeline asynchrone typique d'Octasic

3.3.2 Machine à états finis du jeu de circuit ITC99

L'efficacité du module de test machine à états finis *FSM_test* est vérifié à l'aide du circuit composé d'un générateur d'horloge et d'un module *b01*, *b02* ou *b06* du benchmark ITC99. Le module de test *FSM_test* est intégré dans la boucle d'horloge du générateur et les registres de la FSM sont remplacés par leur équivalent testable (Figure 3.5). Le chronogramme (Figure 3.6) illustre l'exécution d'un test de transition de *launch-on-capture* à l'aide du

module *FSM_test*. On observe que pour une impulsion sur le signal *scan_clk*, le module produit une impulsion de lancement et une de capture sur la sortie *clk_out*.

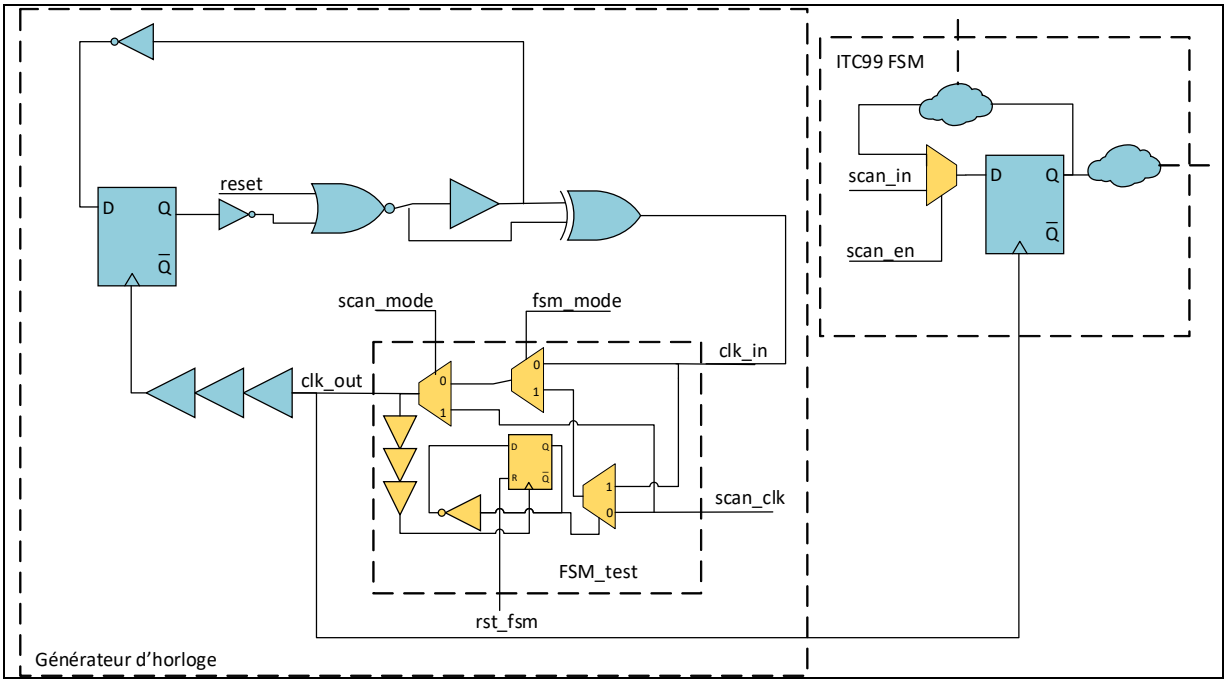

Figure 3.5 Schéma représentant l'intégration des structures de test dans la machine à états finis autoséquencés

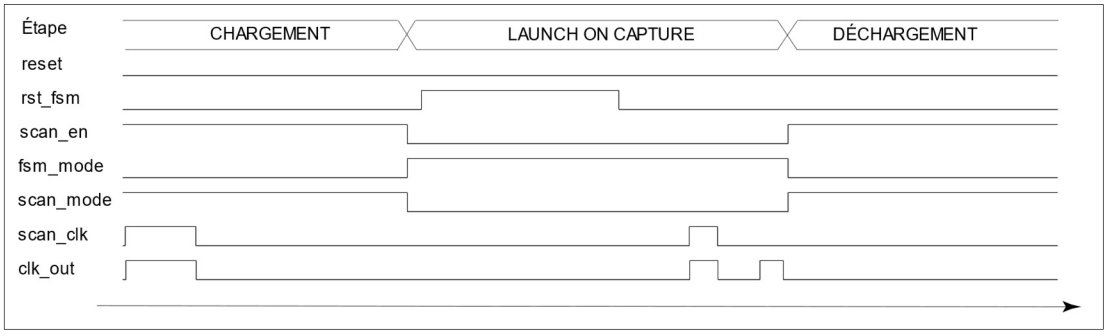

Figure 3.6 Chronogramme tiré de la simulation d'un test *launch-on-capture* à l'aide du module *FSM_test*

3.3.3 Mini-MIPS asynchrone

Le mini-MIPS est un circuit asynchrone complexe. Par conséquent, l'ensemble des chemins de données ne respecte pas entièrement le modèle d'un pipeline d'Octasic typique présenté en revue de littérature. On distingue 8 types de chemin de donnée :

- les chemins de données présents dans les machines à états finis;
- les chemins de données convergents;
- les chemins de données convergents spéciaux ;
- les chemins de données indirectement convergents de catégorie 1;
- les chemins de données indirectement convergents de catégorie 1 spéciaux;
- les chemins de données indirectement convergents de catégorie 2;
- les chemins de donnée dans les pipelines synchrones; et
- les chemins de données sans registre de capture.

Chemins de données présents dans les machines à états finis

Le mini-MIPS contient quelques machines à états finis. Elles sont principalement présentes dans l'unité de calcul et le compteur de programme.

Chemins de données convergents

Les chemins de données convergents *(Converging Datapath - CD)* (Figure 3.7) représentent ce qui se rapproche le plus du cas idéal du pipeline typique de l'architecture d'Octasic. C'est-à-dire que l'horloge de lancement P0 traverse uniquement un *key unit* et une ligne à délai.

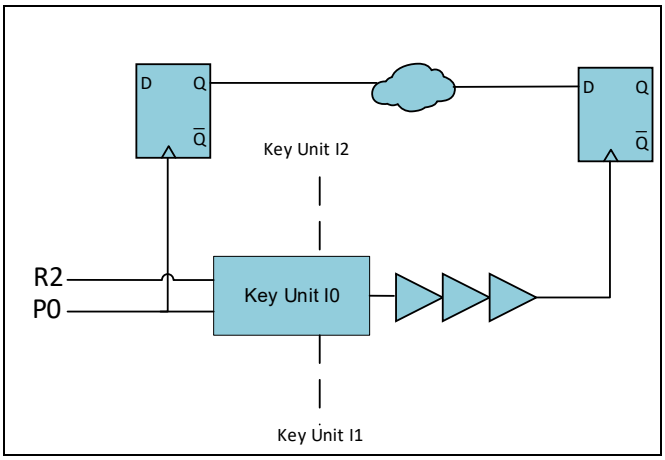

Figure 3.7 Exemple de chemin de données convergent dans le mini-MIPS

Chemins de données convergents spéciaux

Les chemins de données convergents spéciaux *(Special Converging Datapaths - SCD)* (Figure 3.8) sont les mêmes que les chemins de données convergents à l'exception que le registre de lancement se situe en aval d'un *key unit*.

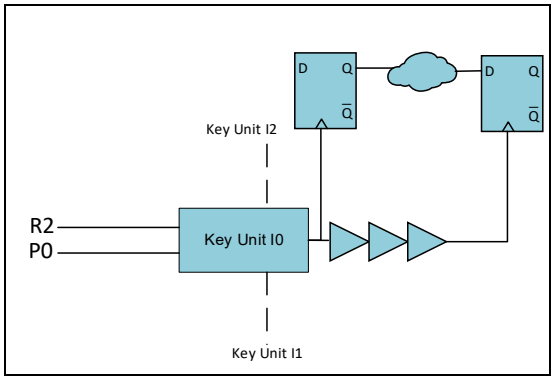

Figure 3.8 Exemple d'un chemin de données convergent spécial

Chemins de données issues de pipelines synchrones

Bien que majoritairement asynchrones, quelques chemins de données issues de pipelines synchrones sont aussi présents.

Chemins de données sans registre de capture

Comme évoqué dans le CHAPITRE 2, à cause de l'extraction des modules contenant des banques de mémoire dans le banc de test, quelques chemins de données sans registre de capture ont été créés.

Chemins de données non directement convergents de catégorie 1

Les chemins de données non directement convergents de catégorie 1 (*Indirectly-converging category 1 datapaths – ICC1D*) respectent les critères suivants:

- plus d'un *key unit* se trouvent dans le chemin d'horloge entre l'impulsion de lancement et celle de capture;
- le premier et dernier *key unit* impliqués dans la séquence nécessaire à la génération de l'horloge de capture par rapport à l'horloge de lancement donnent accès à la même ressource (PC, *Imem*, RF, ALU, *Dmem*). Ils sont donc reliés par le même signal *key*.

Exemple : Les Figure 3.9 et Figure 3.12 contiennent le même ICC1D coloré en rouge. On observe que l'horloge de lancement de ce chemin est R0, et l'horloge de capture est A2. Pour qu'une impulsion soit déclenchée sur l'horloge de capture A2, 5 *key units* se trouvent dans le chemin d'horloge depuis l'horloge de lancement R0. Le premier et dernier *key unit* donnent tous les deux accès à l'ALU, les critères ICC1D sont donc respectés.

Chemins de données non directement convergent de catégorie 1 spéciaux

Les chemins de données non directement convergents de catégorie 1 spéciaux *(Indirectly-converging category special 1 datapaths – ICCS1D)* respecte les critères suivants :

- plus d'un *key unit* se trouvent dans le chemin d'horloge entre l'impulsion de lancement et celle de capture.;
- le premier et dernier *key unit* impliqués dans la séquence nécessaire à la génération de l'horloge de capture par rapport à l'horloge de lancement donnent accès à la même ressource. Ils sont donc reliés par le même signal *key*;
- contrairement au chemin de type ICC1D, le registre de lancement du chemin ICCS1D se situe en aval d'un *key unit* de la même manière que les chemins convergents spéciaux.

Exemple : Deux ICCS1D sont illustrés à la Figure 3.12, on observe que la source de leur horloge de lancement se situe en aval d'un *key unit*.

Chemins de données non directement convergents de catégorie 2

Les chemins de données non directement convergents de catégorie 2 *(Indirectly-converging category 2 datapaths – ICC2D)* respectent les critères suivants :

- plus d'un *key unit* se trouvent dans le chemin d'horloge entre l'impulsion de lancement et celle de capture;
- le premier et dernier *key unit* impliqués dans la séquence nécessaire à la génération de l'horloge de capture par rapport à l'horloge de lancement donnent accès des ressources **différentes**.

Exemple : la Figure 3.10 contient un ICC2D. On observe que l'horloge de lancement est R1 et l'horloge de capture est P2. Le premier *key unit* A1 impliqué donne accès à l'ALU tandis que le dernier *key unit* P2 contrôle le compteur de programme. 6 *key units* se trouvent dans le chemin d'horloge entre l'impulsion de lancement et celle de capture. Les critères ICC2D sont donc respectés.

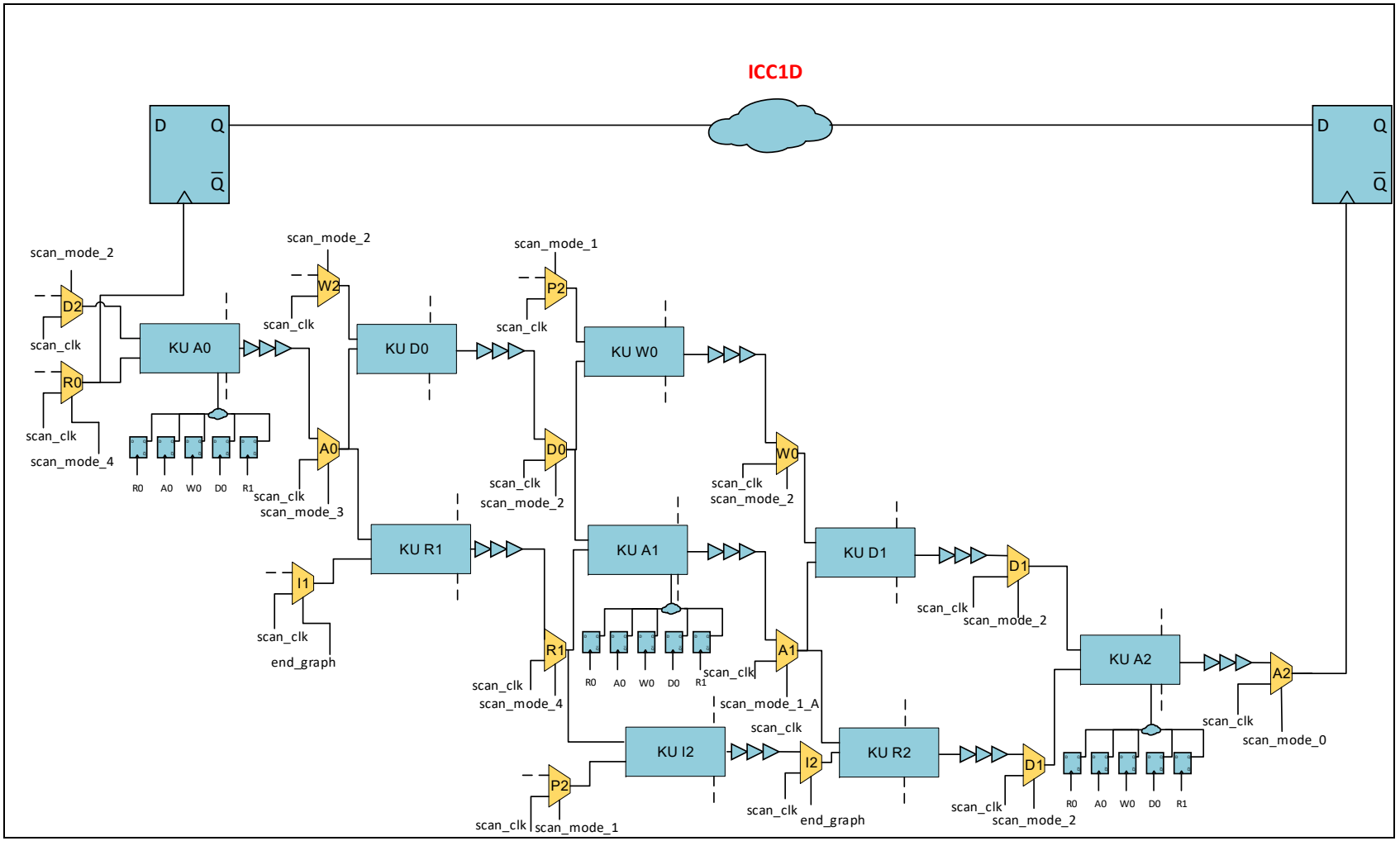

Figure 3.9 Exemple d'un chemin de données de type ICC1D dans le mini-MIPS, les modules *Bypass_KU* et *FSM_test* sont volontairement omis pour ne pas surcharger le schéma

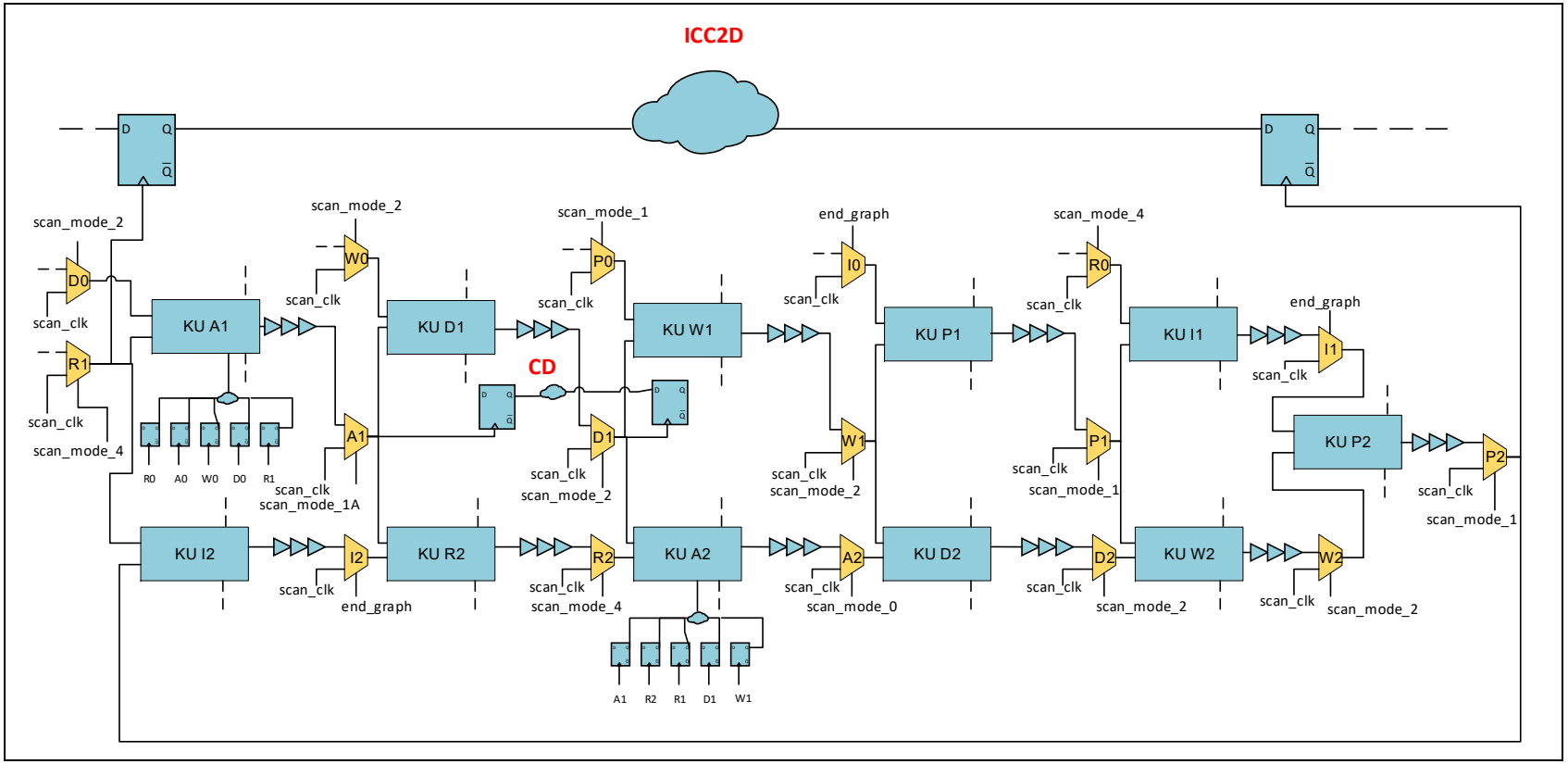

Figure 3.10 Exemple de chemin de données de type ICC2D et CD dans le mini-MIPS, les modules Bypass_KU et FSM_test sont volontairement omis pour ne pas surcharger le schéma

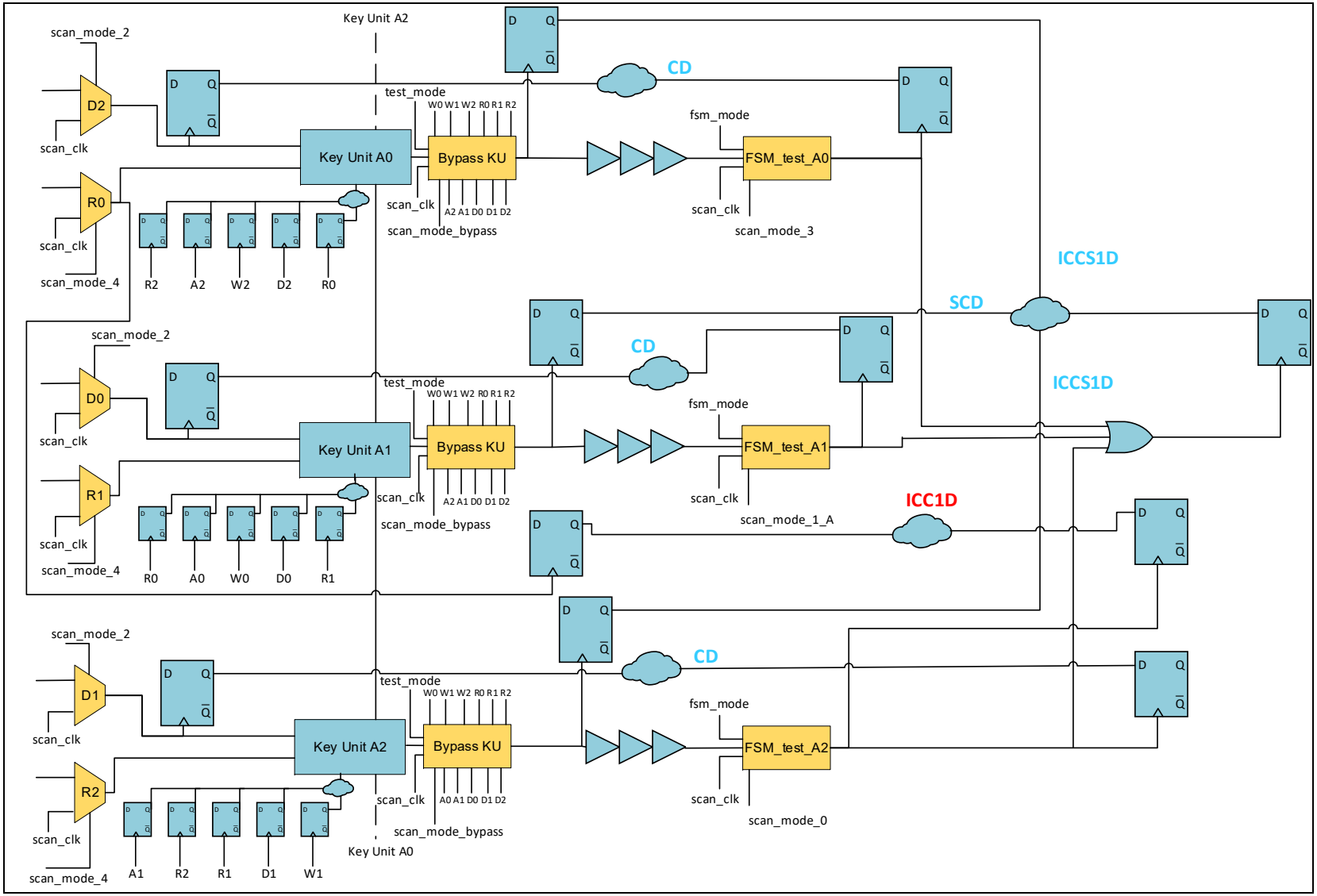

Figure 3.12 Insertion et connexion des structures de test dans le mini-MIPS

3.3.4 Stratégie de test des chemins de données

L'objectif de cette section est de montrer comment les structures de test sont utilisées pour tester les 8 types de chemins de données précédemment définis. Certains détails, comme le choix et la connexion des signaux de type *scan_mode* ou encore les horloges reliées sur les modules *Bypass_KU* seront expliqués dans le CHAPITRE 4 sur l'automatisation du processus de test. La lecture de cette section permet essentiellement de nous apprendre comment sont générées les impulsions de lancement et de capture lors d'un test de transition *launch-on-capture* pour chaque type de chemin de données.

Chemins de données convergents (CD)

Le chemin peut être testé en appliquant la méthode des multiplexeurs de test présenté en section 3.2.1. Par exemple, pour tester le chemin (Figure 3.13) en *launch-on-capture*, le multiplexeur de test P0 doit être configuré en mode de lancement (entrée *scan_clk*) et le multiplexeur I0 est réglé en mode de capture (entrée connectée à la ligne à délai). L'impulsion de lancement provenant de P0 est récupérée par le module de contournement *Bypass_KU* pour se rendre dans le multiplexeur I0 et capturer la donnée.

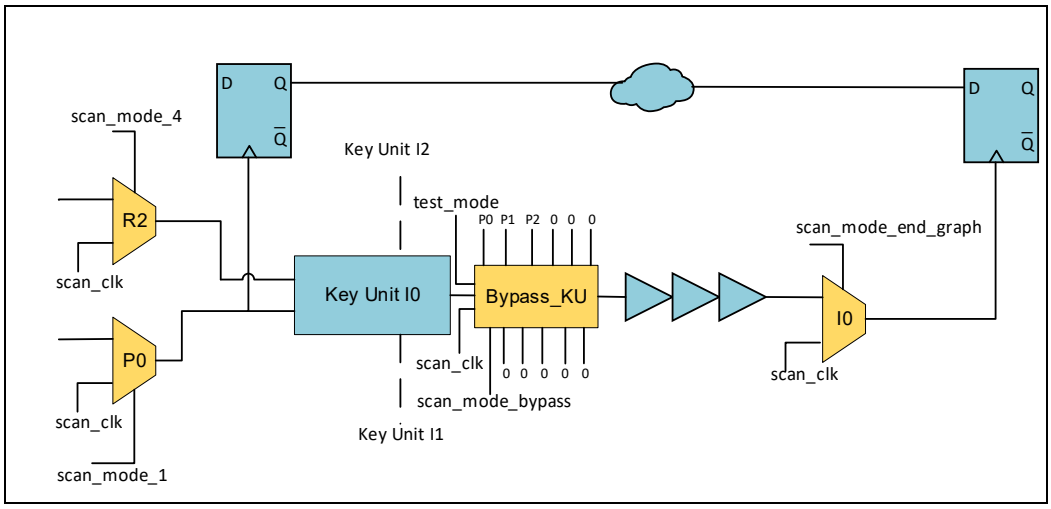

Figure 3.13 Exemple de l'insertion des structures de test dans un chemin de données convergent du mini-MIPS, l'horloge de lancement P0 traverse uniquement un *key unit* et une ligne à délai pour générer l'horloge de capture I0

Chemins de données convergents spéciaux (SCD)

Ce type de chemin est testé en appliquant la même méthode que les *chemins de données convergents*. Dans le mode de fonctionnement normal, l'horloge de lancement est générée par le *key unit*. Or, pendant le mode de test du circuit, le *key unit* ne fonctionne pas et ne génère donc pas d'impulsion. On utilise alors les horloges déjà connectées au module de contournement *Bypass_KU* pour le lancement du chemin de données. En l'occurrence sur la Figure 3.14, P0 est utilisé par l'intermédiaire de *Bypass_KU* pour le lancement des données.

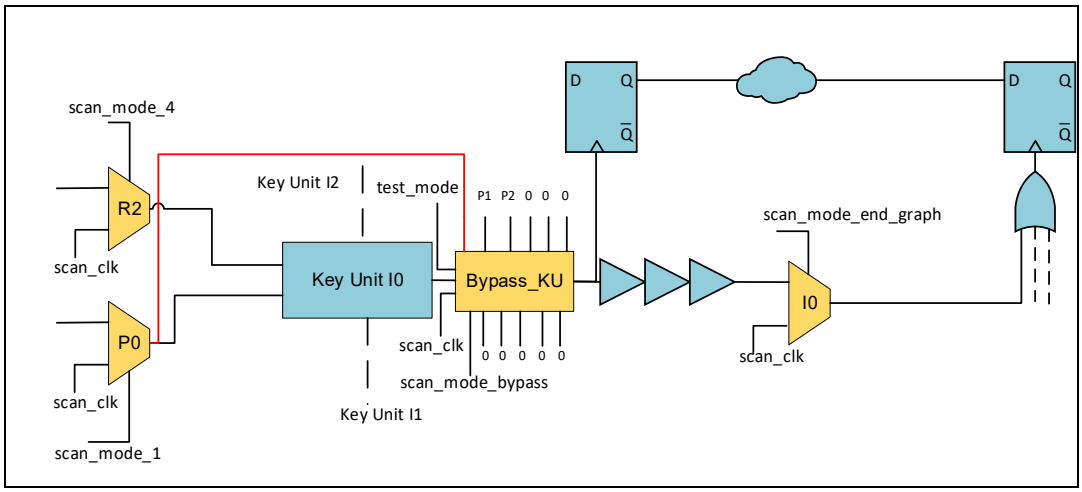

Figure 3.14 Exemple de l'intégration des structures de test dans un chemin de donnée convergent spécial du mini-MIPS, l'horloge P0 est utilisée pour le lancement du chemin de donnée

Chemins de données non directement convergents de catégorie 1 (ICC1D)

Les chemins de type ICC1D sont testés de la même manière que les *chemins convergents*, car ils sont confondus par le processus d'automatisation. L'horloge de lancement d'un tel chemin est donc automatiquement connectée au module de contournement qui lui-même est relié à l'horloge de capture du chemin. Par exemple, dans la Figure 3.12, on peut observer que l'horloge de lancement R0 du chemin ICC1D est connectée au *Bypass_KU* qui génère l'horloge de capture A2.

Chemins de données non directement convergent de catégorie 1 spéciaux (ICCS1D)

Les chemins de type ICCS1D sont aussi testés de la même manière que leurs chemins semblable ICC1D. La seule différence est la topologie du chemin qui prend son horloge de lancement en aval du *key unit*. Comme dans les chemins de données convergents spéciaux, l'horloge de lancement est récupérée à travers le module de contournement *Bypass_KU*.

Chemins de données non directement convergents de catégorie 2 (ICC2D)

Tel que nous le verrons dans le CHAPITRE 4, les chemins de type ICC2D ne sont pas couverts par le processus de testabilité automatisé. Ils impacteront donc forcément négativement notre taux de couverture de panne.

Chemins de données présents dans les machines à états finis

Les chemins de données présents dans les machines à états finis ne peuvent pas être testés en utilisant la méthode des multiplexeurs de test puisque l'horloge de lancement et de capture doit être appliquée sur un seul même nœud. On utilisera plutôt la solution détaillée précédemment du module de test machine à états finis *FSM_test* décrite dans la section 2.2. Le module *FSM_test* devra être intégré dans chaque domaine d'horloge présentant une machine à états finis. On peut observer l'intégration de ces modules dans le mini-MIPS (Figure 3.12).

Chemins de données issues de pipelines synchrones

Ces chemins pourront être testés en appliquant une méthode de *launch-on-capture* standard synchrone, en générant l'impulsion de lancement et de capture depuis le signal *scan_clk*. La procédure sera détaillée dans le CHAPITRE 5. On peut observer ce type de chemin sur la Figure 3.15.

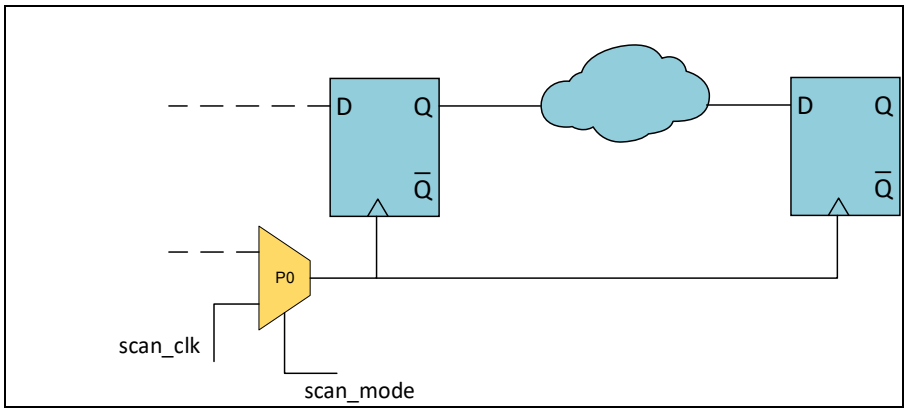

Figure 3.15 Chemin de donnée employant une topologie synchrone dans le mini-MIPS

Chemins de données sans registre de capture

Les chemins de données sans registre de capture ne pourront pas être testés en *launch-on-capture*. On pourra tout de même tester le bon fonctionnement de leurs logiques avec le modèle de test collé-à.

3.3.5 Importance du test à vitesse nominale (*at-speed*)

Tous les chemins de données définis dans la section précédente n'ont pas besoin d'être testés en *launch-on-capture* à vitesse nominale. En effet, certains chemins sont plus critiques que d'autres.

Le test des chemins de données convergents est la priorité du processus de testabilité automatisé. Ce sont les chemins avec l'écart de temps le plus court entre l'impulsion de lancement et celle de capture. Ils sont donc plus susceptibles aux pannes potentielles qui raccourciraient le temps de propagation de l'impulsion de lancement dans le chemin d'horloge. Une telle panne de délai aurait pour conséquence la corruption des signaux dans le chemin de données. La panne réciproque, c'est-à-dire, celle qui aurait tendance à rallonger le

délai dans le chemin d'horloge est moins importante, car elle ne présente aucun risque pour la corruption des données.

Le test de transition des chemins de données de type ICC1D, ICCS1D et ICC2D ne constitue pas une nécessité. En effet, dans ces chemins non directement convergents, la relation entre l'horloge de lancement et de capture implique plus d'un *key unit*. L'horloge de capture est donc le résultat d'une succession d'impulsions d'horloges différentes (Figure 3.12 et Figure 3.13). La différence de temps entre le départ de l'horloge de lancement et l'arrivée de l'horloge de capture est donc forcément très importante. Du fait de la construction du mini-MIPS asynchrone, les chemins de données de type ICC1D, ICCS1D et ICC2D ne sont donc pas critiques.

Dans la pratique, nous verrons qu'à cause de l'algorithme de construction de dépendance des horloges que nous détaillerons dans le CHAPITRE 4, les chemins de données convergents et ICC1D sont confondus par le programme. Par exemple, sur la Figure 3.12, l'horloge de lancement R0 se retrouve connectée par l'algorithme en entrée du module de contournement du *key unit* de l'horloge de capture A2. Il n'y a donc qu'une ligne à délai impliqué dans le retardement de l'horloge de lancement. Dans le cas réaliste du fonctionnement normal du circuit représenté, l'horloge de lancement est au moins retardée par 5 lignes à délai de 5 *key units* (ex. : A0-R1-A1-R2-A2). Les chemins de données ICC1D seront donc testés plus vite que leur vitesse nominale. C'est aussi le cas des chemins de données dans les machines à états finis où uniquement une ligne à délai est impliquée en mode de test. Dans le cas du fonctionnement normal, au moins 2 lignes à délai sont impliquées dans la génération successive des horloges de lancement et de capture. Par exemple, pour la machine à états finis présente dans l'ALU, une des séquences les plus courtes d'activation des *key units* est A0-R1-A1 (Figure 3.9). On verra dans le CHAPITRE 6 que ces tests des chemins non convergents plus rapides que la vitesse nominale mènent à des problèmes *d'over-testing*, c'est-à-dire qu'on peut considérer un circuit comme défectueux alors qu'il ne l'est pas.

3.4 Conclusion

Ce chapitre a été l'occasion de présenter la stratégie de test mise en place pour chaque circuit du banc d'essai. Certains détails restent inconnus, car ils sont relatifs à l'automatisation du processus de test et seront présentés dans le prochain chapitre qui lui est consacré. On a pu constater grâce au mini-MIPS qu'il est compliqué de construire un circuit asynchrone complexe qui respecte entièrement la topologie des circuits d'Octasic. Avec 7 autres types de chemin de données, les chemins convergents sont en effet loin de couvrir toutes les possibilités de parcours des données dans le mini-MIPS. Toutes ces nouvelles topologies de chemin n'ont pas été prévues à la base par la méthode de test initial. Elles compliquent donc notre stratégie de test et nous imposent de mettre en place de nouvelles structures particulières au circuit comme le module de contournement des *key unit*. Ces mesures particulières compliquent notre tentative de déterminer un flot de testabilité unique à la topologie de conception d'Octasic. Un risque se profile, c'est celui de créer un flot automatisé de testabilité exclusif au mini-MIPS et inemployable sur d'autres architectures de circuits. Avec l'objectif d'avoir un processus de testabilité le plus standardisé possible, on devra donc s'assurer de séparer les mesures particulières au mini-MIPS des actions nécessaires à tous les circuits employant la topologie d'Octasic.

CHAPITRE 4

AUTOMATISATION DE L'INSERTION DES STRUCTURES DE TEST SPÉCIFIQUES

4.1 Introduction

Ce chapitre présente l'automatisation d'une partie du flot de conception pour la testabilité, à savoir l'intégration des structures de test spécifiques à l'architecture d'Octasic vue dans le CHAPITRE 3. Rappelons que l'intégration de ces structures spécifiques, qui inclut des structures génériques (pour tous les circuits de type Octasic) et des structures plus spécialisées (pour certains types de circuits) s'ajoute à celle des structures de test conventionnelles utilisées dans les designs synchrones.

La première partie de ce chapitre apporte une vue générale sur le flot complet de conception pour la testabilité adaptée aux circuits de type Octasic, à l'intérieur duquel on retrouve notamment l'insertion des types de structures de test précédemment mentionnés. La deuxième partie décrit la première phase du flot de testabilité qui consiste à intégrer les structures conventionnelles (i.e. les chaînes de balayage) et les structures génériques (i.e., les multiplexeurs de test). La troisième partie se concentre sur la plus grosse contribution apportée par ce mémoire, c'est-à-dire le programme d'analyse et connexion des signaux de contrôle des multiplexeurs de test. Celle-ci sera découpée en quatre sous-sections. Dans l'objectif d'avoir tous les outils à la compréhension des algorithmes, la première sous-section nous fera un rappel sur la théorie des graphes. La deuxième sous-section présentera l'utilisation du programme et quelques détails d'implémentation. Enfin, la troisième sous-section permettra un survol du fonctionnement du programme. Pour finir, on détaillera dans la dernière sous-section les algorithmes les plus représentatifs et importants du programme.

4.2 Flot complet de conception pour la testabilité

L'automatisation du flot de conception pour la testabilité a été réalisée avec un ensemble de scripts qui utilise des outils de développement commerciaux et un programme entièrement développé pour l'occasion. Les différents scripts utilisent tous le langage script TCL (Tool Command Language), ils servent à contrôler les deux principaux outils utilisés dans ce flot de conception pour la testabilité : Genus de Cadence qui est notre outil de synthèse et Tessent de Mentor Graphics qui regroupe toutes les fonctionnalités relatives au test de circuit intégré.

On peut découper ce flot de testabilité en 4 phases illustrées à la Figure 4.1:

1. L'insertion des structures de test conventionnelles et génériques.
2. La détermination des signaux de contrôle de test.
3. L'insertion des structures de test spécialisées.
4. La génération automatisée des vecteurs de test.

Notons que ce chapitre se concentre sur les trois premières phases et que la quatrième fait l'objet du chapitre suivant.

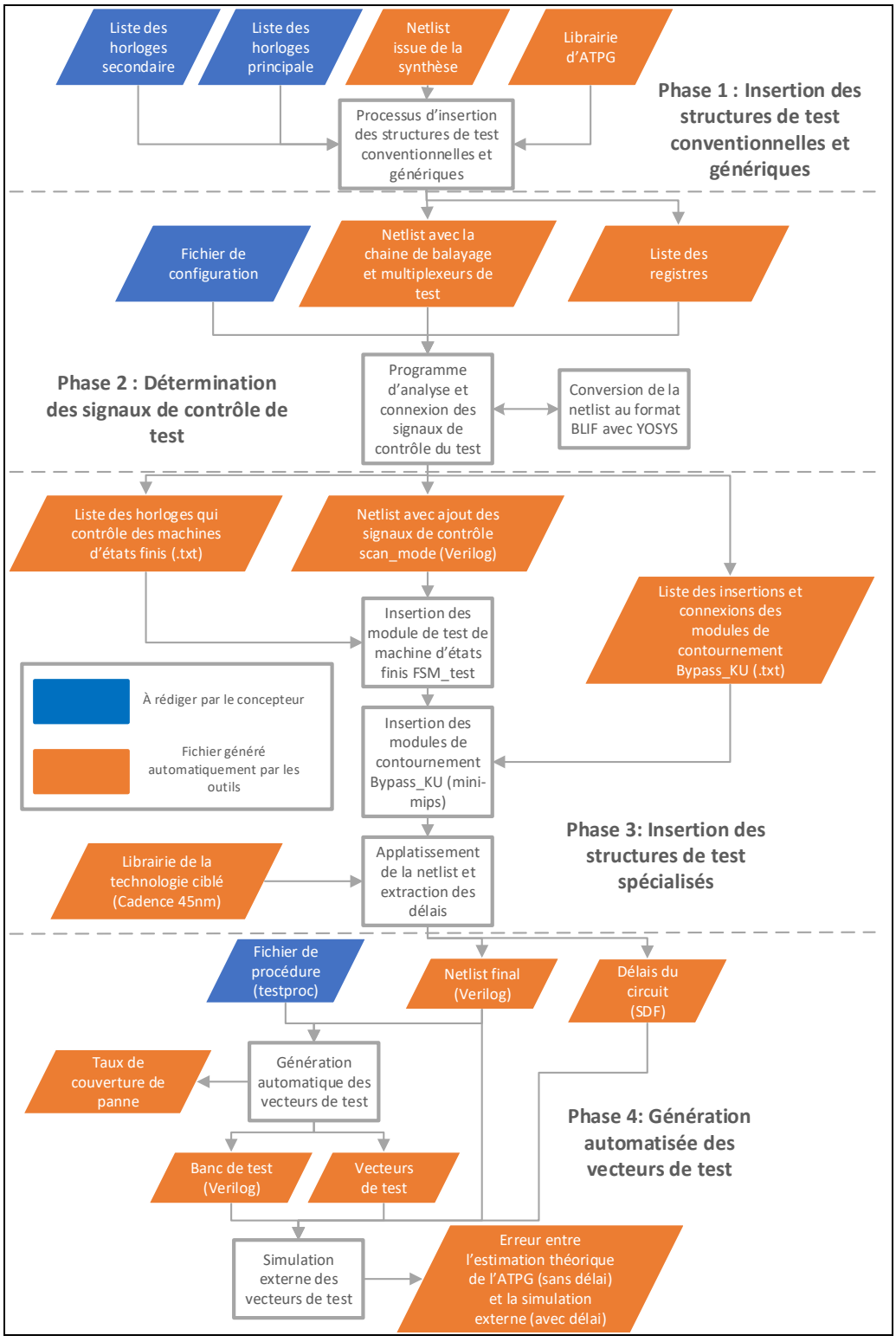

Figure 4.1 Diagramme représentant le processus de testabilité automatisé présenté dans ce mémoire

4.3 Insertion des structures de test conventionnelles et génériques

L'insertion des structures de test conventionnelles et génériques est la première phase après avoir obtenu la netlist du circuit grâce à un outil de synthèse, Genus dans notre cas. Elle consiste à insérer la (les) chaîne(s) de balayage ainsi que les multiplexeurs de test dans le circuit initial. Ce processus est réalisé à l'aide d'un script exécuté par Tessent et disponible en ANNEXE I. Par opposition, le mini-MIPS testé dans ce mémoire se verra ajouter plus tard dans le flot d'autres structures de test plus spécialisées en plus de ces structures conventionnelles et génériques.

Le processus d'insertion est décrit sur la Figure 4.2. Quatre fichiers en entrée sont nécessaires pour générer notre nouvelle netlist : la liste des horloges principales, la liste des horloges secondaires, la netlist générée par la synthèse et la libraire d'ATPG de la technologie ciblée.
Le concepteur doit ici identifier les horloges du circuit et les répartir en deux listes : la liste des horloges principales et la liste des horloges secondaire.

La liste des horloges principale correspond aux horloges du circuit où seront effectuées les impulsions de lancement et de capture lors de nos tests *launch-on-capture*. C'est aussi grâce à cette même liste que les nœuds d'insertion des multiplexeurs de test sont localisés.

La liste des horloges secondaires a été ajoutée lorsque les résultats de l'analyse de la couverture de panne ont montré que tous les chemins de données n'étaient pas couverts. Ces horloges correspondent à la sortie des *key units* dans le mini-MIPS (Figure 4.3). Cette liste doit donc comporter toutes les horloges où l'on ne veut pas que des multiplexeurs de test soient insérés. Le concepteur devra tout de même penser à ajouter une autre structure spécialisée pour contrôler les registres de balayage associés à cette horloge secondaire, dans notre cas, c'est le module de contournement *Bypass_KU* qui joue ce rôle.

Les autres fichiers, c'est-à-dire, la netlist d'entrée et la librairie d'ATPG sont générées automatiquement par les outils de développement. Il faut noter que Tessent, le logiciel de

testabilité utilisé dans ce mémoire, offre un outil de conversion appelé *libcomp* pour créer la librairie pour ATPG à partir de la librairie de la technologie cible. C'est de cette manière que nous avons procédé pour obtenir cette libraire.

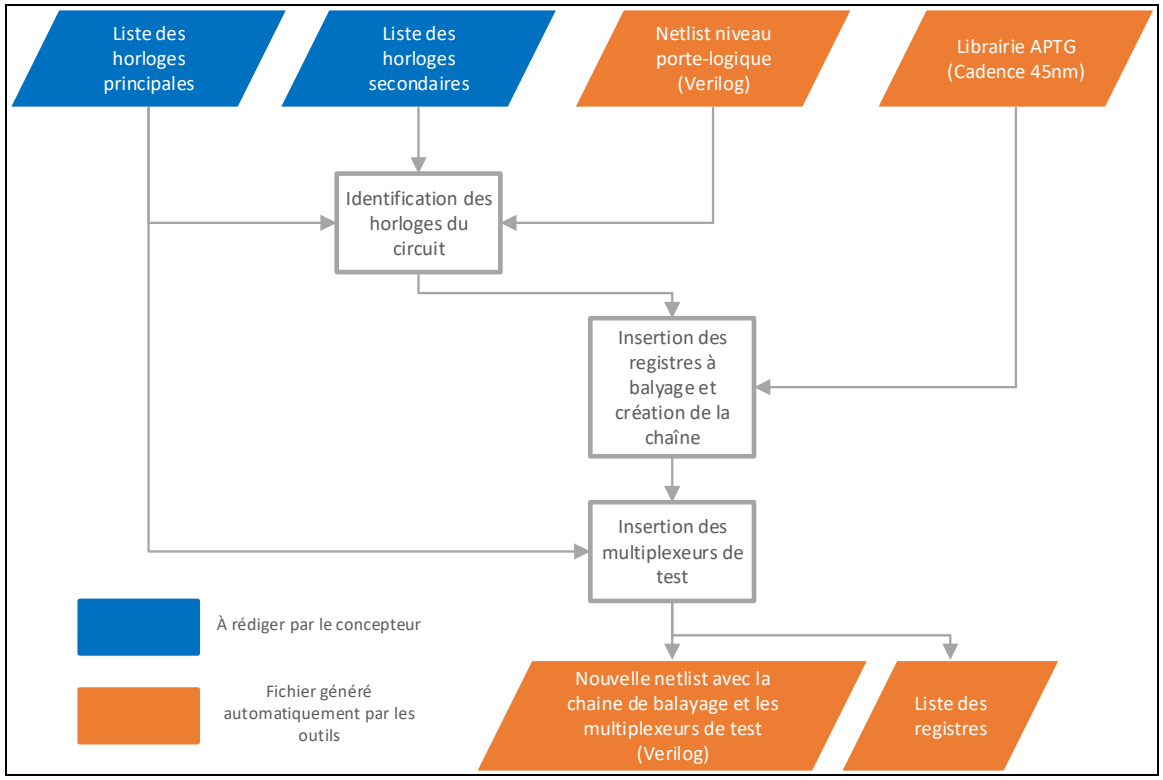

Figure 4.2 Diagramme représentant le processus d'insertion des structures génériques

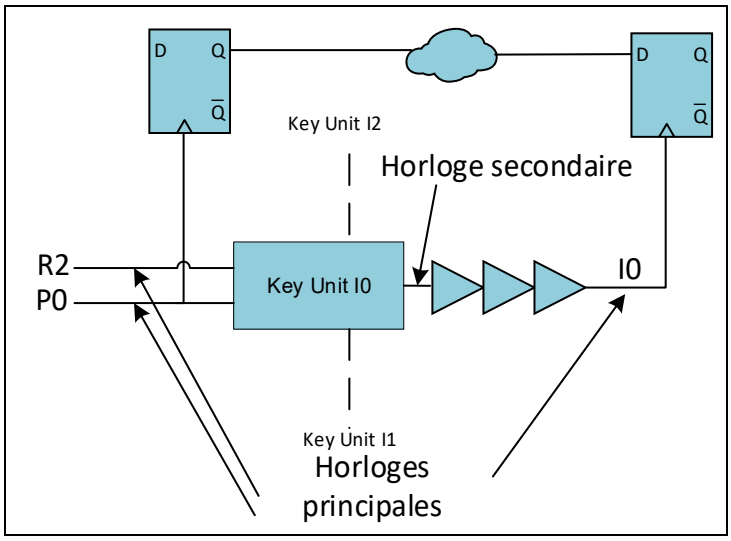

Figure 4.3 Schéma représentant un chemin de données convergent du mini-MIPS, on y détaille les horloges principales R2,P0 et IO et l'horloge secondaire en sortie du key unit

4.4 Détermination des signaux de contrôle de test

Cette deuxième phase est celle qui a demandé le plus d'efforts dans ce mémoire. Les multiplexeurs de test insérés dans la première phase sont tous contrôlés par des signaux qu'on appelle *scan_mode.* Leur but est de sélectionner la source de l'horloge propagée par les multiplexeurs de test qui provient soit du signal d'horloge globale *scan_clk* contrôlée par le testeur ou une ligne à délai du circuit. Comme expliqué en revue de littérature et dans le CHAPITRE 3, ces signaux *scan_mode* sur des multiplexeurs de test successifs doivent être différents pour permettre des tests de transition *launch-on-capture*. Le rôle de cette deuxième phase est donc de déterminer le nombre nécessaire de signaux de type *scan_mode* pour que tous les multiplexeurs de test successifs dans le circuit soient contrôlés par des signaux différents. Ce processus est entièrement réalisé par un programme développé pour l'occasion. Le logiciel réalise la détermination du nombre de signaux de type *scan_mode* et la connexion de ceux-ci. De plus, il permet la préparation de l'insertion des structures de test spécialisées dans la 3ème phase.

Dans l'ordre, la section présente le programme d'analyse et de connexions des signaux de contrôle des multiplexeurs de test. Les algorithmes utilisés et développés réalisent en majorité des opérations sur des graphes, c'est pourquoi on commencera cette section par un rappel sur la théorie des graphes. À la suite de ce rappel, on présentera le programme en divulguant quelques détails d'implémentation, et son mode d'utilisation. La section 4.4.3 sera l'occasion d'offrir un survol général du fonctionnement du programme tandis que le fonctionnement détaillé des algorithmes les plus importants sera présenté en 4.4.4.

4.4.1 Rappel sur la théorie des graphes

Selon (Cogis & Robert, 2003), un graphe est constitué d'un ensemble X et d'un ensemble E de paires d'éléments de X. Les éléments X correspondent au sommet du graphe G tandis que ceux de E sont les arêtes du graphe.

$$G = (X, E) \tag{4.1}$$

Le graphe non orienté représenté en Figure 4.4 est défini par l'ensemble de sommets X de l'équation 4.2 et l'ensemble d'arêtes E de l'équation 4.3.

$$X = \{1,2,3,4,5,6,7\} \tag{4.2}$$

$$E = \{\{1,5\},\{1,6\},\{1,7\},\{2,4\},\{2,3\},\{3,4\},\{4,7\},\{4,5\},\{7,1\}\} \tag{4.3}$$

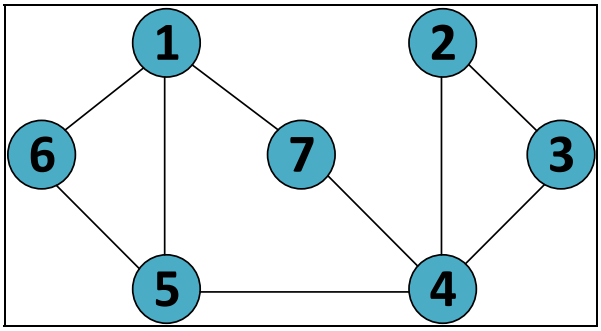

Figure 4.4 Graphe non orienté

L'**ordre** d'un graphe correspond à son nombre de sommets X, ainsi les graphes représentés en Figure 4.4 et Figure 4.5 sont d'ordre 7. On appelle **degré** le nombre de voisins d'un sommet, par exemple le sommet 1 est de degré 3. Les graphes peuvent être **non orientés** (Figure 4.4) ou **orientés** (Figure 4.5). Un graphe est orienté si les arrêtes possèdent une orientation. Un graphe est dit **cyclique** si l'on est capable d'élaborer un chemin qui revient au sommet de départ. On parlera plutôt de **circuit** dans le cas d'un graphe orienté et de **cycle** pour un graphe non orienté. Par exemple le chemin composé par les sommets {1,6,5,4,7} forme un circuit dans le graphe Figure 4.5 et un cycle dans le graphe Figure 4.4. Le graphe est dit **acyclique** s'il ne contient pas de cycle/circuit.

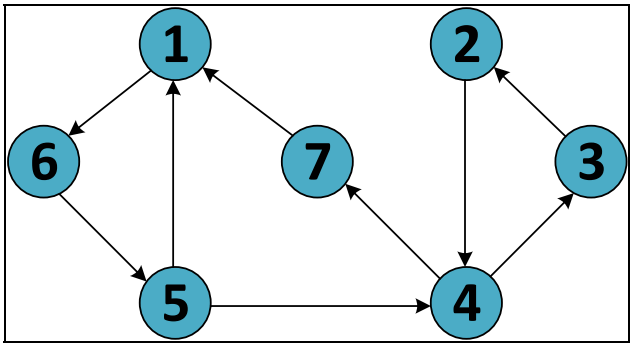

Figure 4.5 Graphe orienté

4.4.2 Présentation et détail d'implémentation

Le programme d'analyse et connexions des signaux de contrôle de test a été développé en C++ et a nécessité environ 3500 lignes de codes sans compter les lignes vides. De plus, plusieurs librairies et logiciels dont nous ne sommes pas les auteurs ont été utilisés pour développer ce programme.

Le programme utilise sa propre méthode de représentation de graphe, mais s'aide de la librairie de Stanford SNAP (Stanford Network Analysis Platform) (Leskovec, 2016) pour les convertir dans un format visualisable.

La visualisation des graphes a été réalisée avec le logiciel open source Graphviz (John Ellson, 2001) qui a été très utile lors du déverminage des algorithmes.

L'algorithme de construction du graphe a été facilité par l'utilisation de YOSYS (Yosys SYnthesis Suite) (Wolf, 2012). Yosys nous permet de convertir une netlist du format verilog au format BLIF (Berkley Logic Interchange Format). Tout comme les netlists en verilog, BLIF décrit un assemblage de cellules standard qui constitue un circuit. La différence est que les connexions d'une cellule sont décrites sur une seule ligne tandis que plusieurs lignes peuvent être utilisées en verilog. Il y a aussi beaucoup moins de caractères spéciaux et le langage utilisé est simplifié au maximum. Toutes ces caractéristiques en font un format de fichier très facile à lire et cela nous a sauvé beaucoup de temps dans le développement de l'algorithme de construction de graphe.

Pour utiliser notre programme, il suffit de lancer l'exécutable tout en ayant préalablement rempli le fichier de configuration (Figure 4.6).

```
Path_blif_in: <PATHS>\<filename>.blif
Path_verilog_in: <PATHS>\<filename>.blif
Path_verilog_out: <PATHS>\<filename>.blif
Path_register: <PATHS>\<filename>.txt
Or_test_mode_connection_report: <PATHS>\<filename>.txt
Path_FSM_module: <PATHS>\<filename>.txt
Port_Interpreted_As_Output: Y Q
Scan_Clock_Signal_Name: scan_clk
Scan_Enable_Signal_Name: scan_en
Test_Mux_Gate_Name: MX2X1
Scan_Mode_Name: scan_m0
Testpoints_signals: token_in testp_en
Delay_cell_name: DLY4X1
```

Figure 4.6 Exemple d'un fichier de configuration pour l'outil

On décrit ci-dessous la signification des différents paramètres du fichier de configuration :

Path_blif_in, Path_verilog, Path_verilog_out, Path_register, Or_test_mode_connection_report, Path_FSM_module: Ce sont les emplacements des fichiers d'entrée et sortie du programme. Correspondent aux fichiers permettant de connecter les modules de contournement des *key units, Bypass_KU* ainsi que les modules de test des machines à états finis *FSM_test.*

Port_Interpreted_As_Output: Les ports inscrits à la suite de ce paramètre seront considérés comme des sorties. C'est grâce à cette liste que l'algorithme de construction de graphe est capable de donner une orientation aux arêtes.
Scan_Clock_Signal_Name: Corresponds au nom du signal d'horloge qui contrôle le chargement des vecteurs dans les registres à balayage.
Scan_Enable_Signal_Name: Corresponds au nom du signal qui permet d'activer le mode de chargement dans les registres de la chaîne de balayage.
Test_Mux_Gate_name: On entre ici le nom de la cellule utilisée pour les multiplexeurs de test.
Scan_Mode_Name: C'est le nom du signal de contrôle connecté à tous les multiplexeurs de test.
Testpoints_signals: On inscrit ici tous les noms des entrées relatives aux points de test
Delay_cell_name: On entre ici le nom de la cellule utilisé pour les lignes à délai principales du circuit.

4.4.3 Fonctionnement général du programme (haut niveau)

Cette sous-section est l'occasion de décrire le fonctionnement général du programme, celui-ci est illustré par la Figure 4.7. On note que les étapes les plus importantes seront décrites dans la section 4.4.4 sur le fonctionnement détaillé du programme.

La première étape du programme est la **conversion de la netlist au format BLIF**. Comme expliqué dans la section 4.4.2, la conversion de la netlist du format verilog au format BLIF permet de simplifier la lecture du fichier. Cette étape est donc suivie de l'algorithme de **construction du graphe** dont l'objectif est de représenter le circuit par un graphe cyclique orienté. Ce graphe sera la base de la majorité des analyses réalisées dans le reste du programme. Notamment la **détection des multiplexeurs de test** dont le rôle est d'identifier et lister tous les multiplexeurs de test insérés dans la phase 1 du flot de testabilité. Toujours dans un objectif de préparation, l'algorithme **d'association des lignes à délai avec les multiplexeurs de test** permet d'associer chaque entrée de ligne à délai à un multiplexeur de

test. Arrive enfin l'**analyse des domaines d'horloges**, dont l'objectif est l'**attribution des domaines d'horloges à chaque registre** du circuit. Pour y arriver, le programme doit d'abord réaliser le **calcul de distance entre les registres et les multiplexeurs de test.** S'en suit l'**analyse de la dépendance des horloges**, son objectif est double : définir la relation entre les horloges et déterminer quelles horloges contrôlent les machines à états finis. Pour ce faire, le programme procède à la **contraction du graphe** en gardant uniquement les multiplexeurs de test dans celui-ci. Ce graphe est ensuite utilisé pour la **définition de la dépendance des horloges.** Enfin, les boucles de rétroaction sur les sommets des multiplexeurs de test permettent la **recherche de machine à états finis.** Le programme se poursuit par la **recherche des chemins de données convergents** qui passent d'abord par une nouvelle étape de **contraction du graphe** initial, cette fois-ci en gardant uniquement les multiplexeurs de test et les registres. Toutes ces différentes étapes parcourues nous mènent à la réalisation de l'objectif principal de ce programme, c'est la détermination du nombre et la connexion des signaux de contrôle des multiplexeurs de test. Cet objectif est obtenu avec l'algorithme de **coloration de graphe.** Enfin, le programme procède à la préparation de l'insertion des modules nécessaire pour le mini-MIPS avec l'étape **de préparation de la connexion des modules *Bypass_KU***. Le programme se conclut par **la génération de la netlist finale et des rapports de *FSM_test* et *Bypass_KU*.** Le rôle de cette dernière étape est de fournir les résultats calculés par le programme. Ainsi une nouvelle netlist est générée. Celle-ci comprend toutes les nouvelles entrées de type *scan_mode* et leurs connexions aux multiplexeurs de test. De plus, deux rapports qui seront utiles pour la suite sont générés. Un premier rapport liste les domaines d'horloges qui comprennent des machines à états finis. Un deuxième rapport donne la liste des connexions à réaliser pour l'insertion des modules de contournement des *Bypass_KU*.

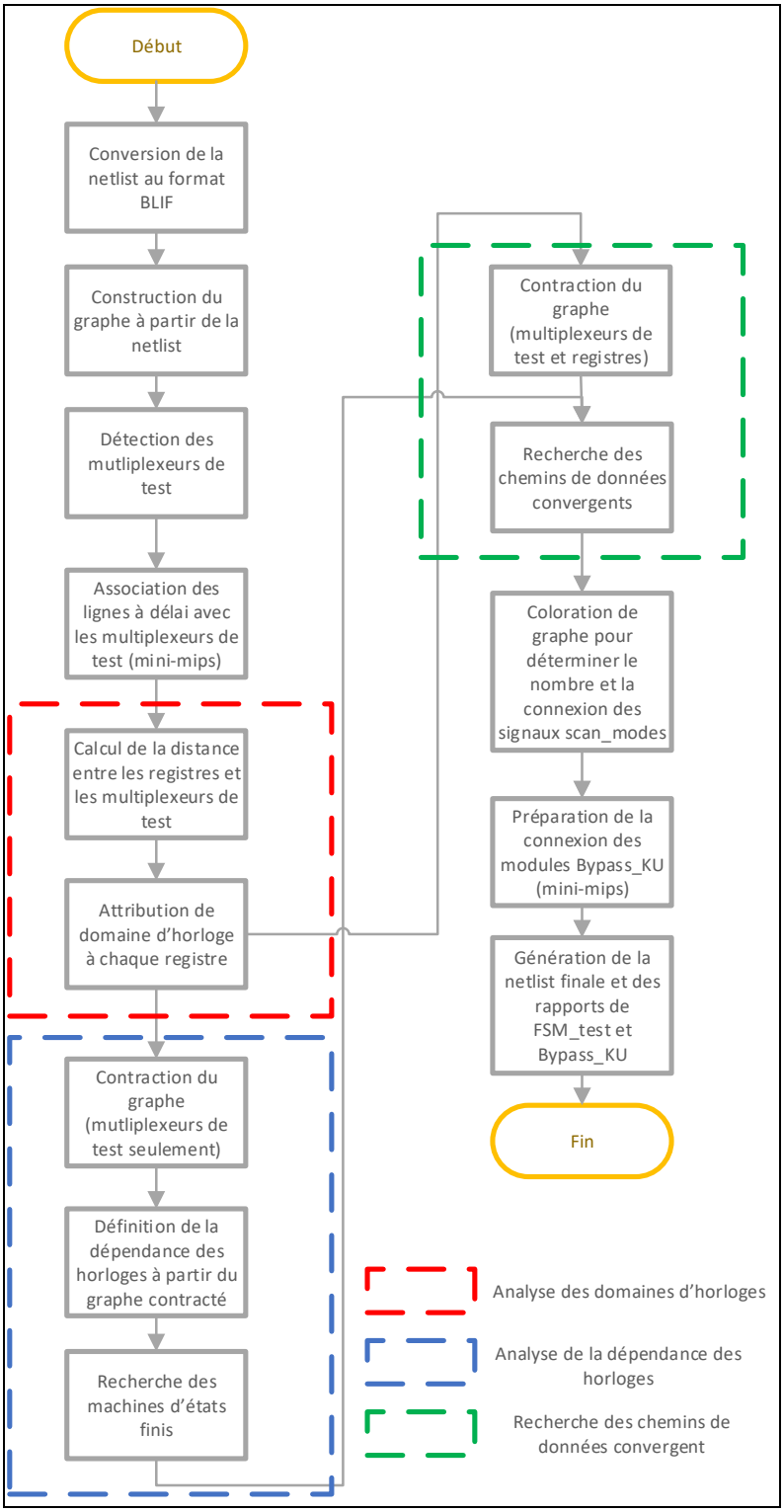

Figure 4.7 Diagramme représentant les étapes de fonctionnement du programme d'analyse et connexion des signaux de contrôle de test

4.4.4 Fonctionnement détaillé du programme

La section suivante présente le fonctionnement détaillé du programme. On y présente seulement les parties jugées les plus complexes et représentatives du programme qui ont été introduites dans la section précédente. Les algorithmes sont simplifiés pour une meilleure compréhension et représentés sous forme de pseudo-code. En effet, celui-ci diffère parfois du code source souvent plus compliqué en raison de détails techniques propre au C++.

4.4.4.1 Conversion de la netlist d'entrée en graphe orienté

Tous les algorithmes du programme manipulent des graphes pour obtenir leurs résultats. La première étape du programme consiste donc à construire un graphe orienté qui représente le circuit. Comme détaillé dans la section 4.4.2, la netlist est convertie à partir de Yosys au format BLIF. Le programme procède alors à la lecture de la netlist transformée pour construire le graphe.

Dans cet algorithme détaillé dans le diagramme Figure 4.10, chaque porte, registre ou fil dans la netlist est converti en sommet dans un graphe. Si l'on prend l'exemple d'un chemin convergent représenté en Figure 4.8, on obtient le graphe orienté en Figure 4.9.

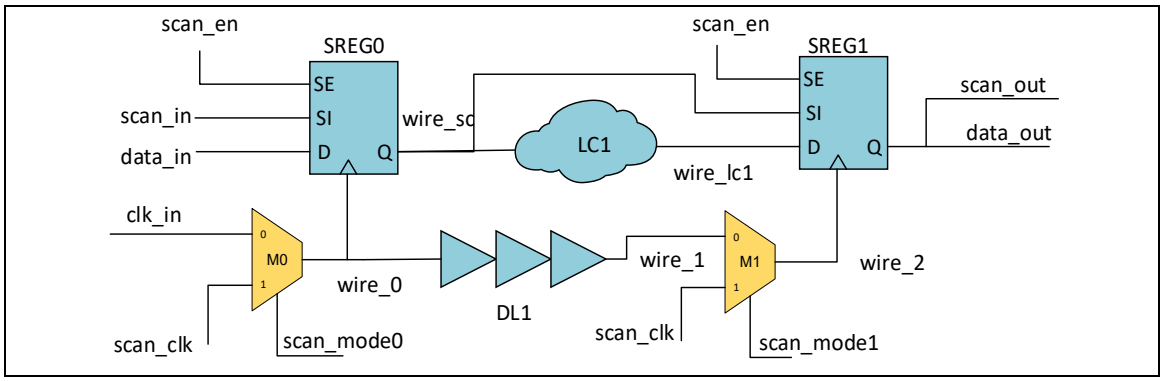

Figure 4.8 Schéma représentant un chemin convergent avec ses structures de test

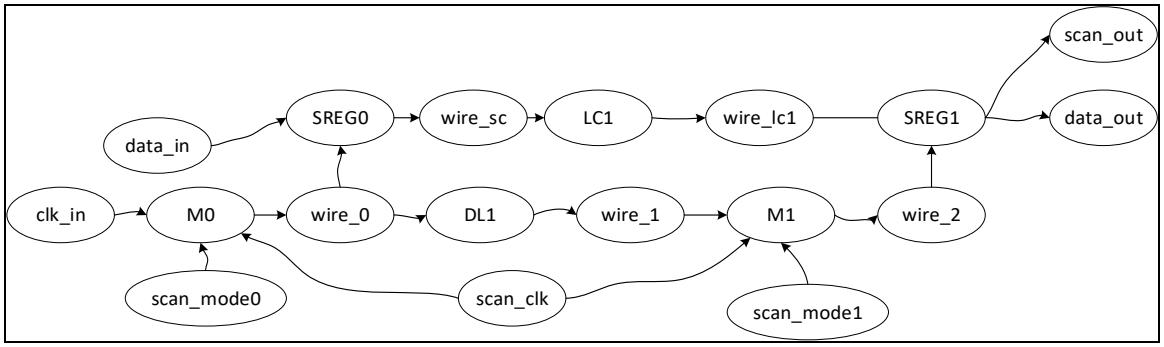

Figure 4.9 Représentation en graphe du schéma du chemin convergent de la Figure 4.8

On notera que seules les connexions dites « standard » sont représentées dans le graphe. C'est-à-dire qu'on évite de représenter les connexions de la chaîne de balayage, car celle-ci pourrait affecter l'analyse du programme et les résultats de nos algorithmes. Pour ce faire, seuls quatre ports sont considérés comme valides quand le programme rencontre un registre, ce sont : Q, QN, D et CK. Ainsi les connexions sur les autres ports comme SI (scan input) ou RN (reset negative) par exemple ne sont pas pris en compte. Enfin, la direction des arêtes est définie par la direction des ports des cellules standards. Pour connaitre cette direction, l'algorithme se base sur la liste des ports considérés comme des sorties dans le fichier de configuration. Les ports qui ne sont pas dans cette liste sont interprétés comme des entrées. Enfin les algorithmes qui suivent cette première étape de construction de graphe nécessitent de pouvoir identifier les registres des autres de cellules standards. Pour ce faire, le programme utilise la liste des registres générée par Tessent pour les identifier lors de la construction du graphe.

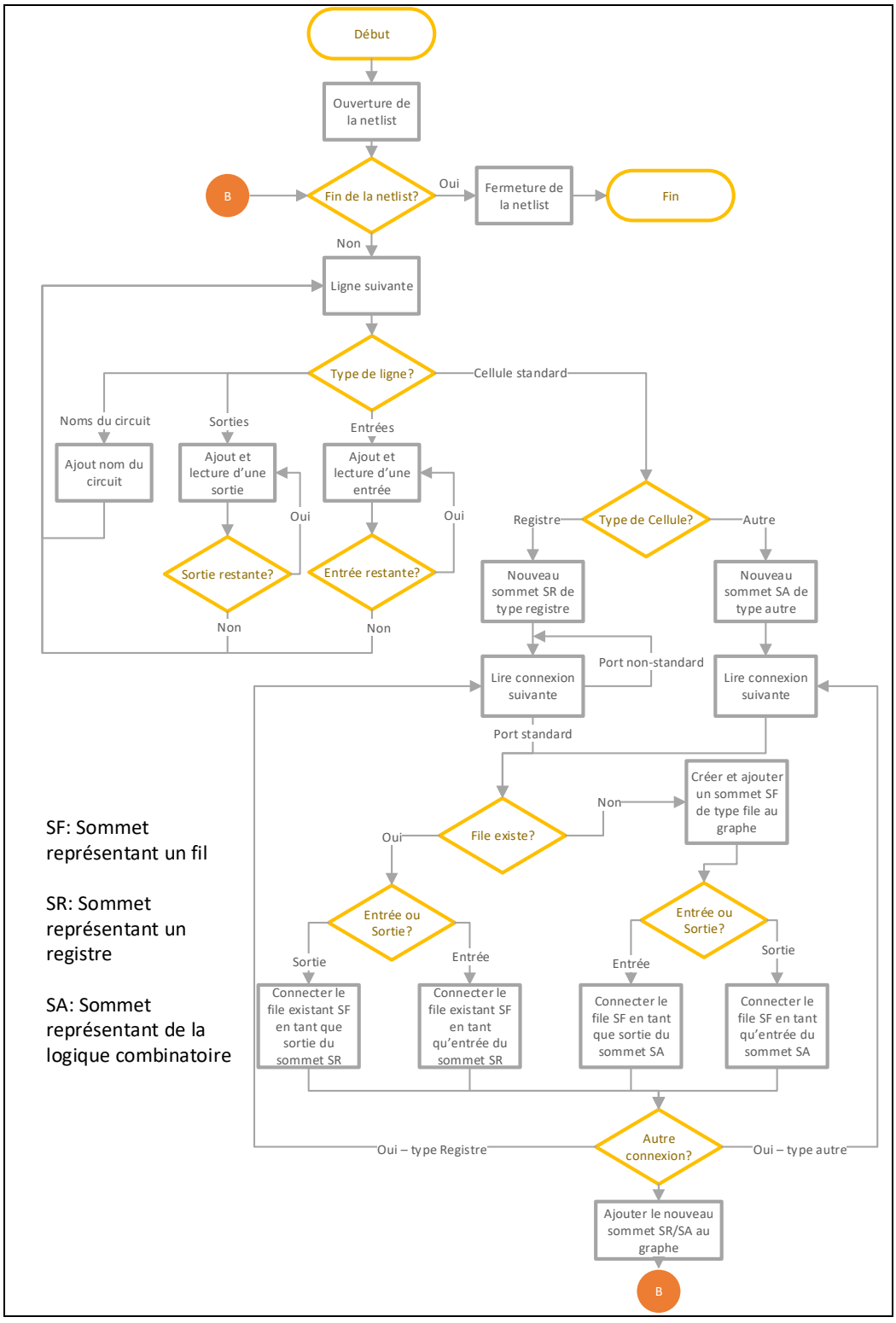

Figure 4.10 Diagramme représentant le fonctionnement de l'algorithme de construction de graphe

4.4.4.2 Détection du réseau de multiplexeur de test

L'objectif de cette étape est de détecter tous les multiplexeurs de test qui ont été insérés dans le circuit. Au terme de l'exécution de cet algorithme, le programme dispose d'une liste qui pointe vers tous les multiplexeurs de test du graphe construit dans l'étape précédente. Cette liste sera utilisée par la plupart des prochaines étapes de notre programme. Cette étape de détection est cruciale puisque la sortie de ces multiplexeurs représente les divers domaines d'horloge du circuit. Dans la suite du chapitre, on confond donc souvent les termes « multiplexeur de test » et « domaine d'horloge ».

Le fonctionnement de l'algorithme est simple puisqu'il utilise le principe bien connu du parcours en largeur (Algorithme 4.1). Pour chaque sommet visité, on l'ajoute dans la liste des multiplexeurs de test si celui-ci est détecté en tant que tel. Pour être considéré comme un multiplexeur de test, le sommet doit répondre à trois critères qui sont :

1. Le sommet doit avoir 3 voisins d'entrée (*scan_clk*, *scan_mode* et l'entrée de la ligne à délai) (Figure 4.11).
2. Un de ces voisins doit être le sommet qui correspond à l'horloge de balayage *scan_clock* renseigné dans le fichier de configuration.
3. Le nom de la cellule standard utilisé doit correspondre à celle renseignée dans le fichier de configuration, *Test_MUX_Gate_Name*.

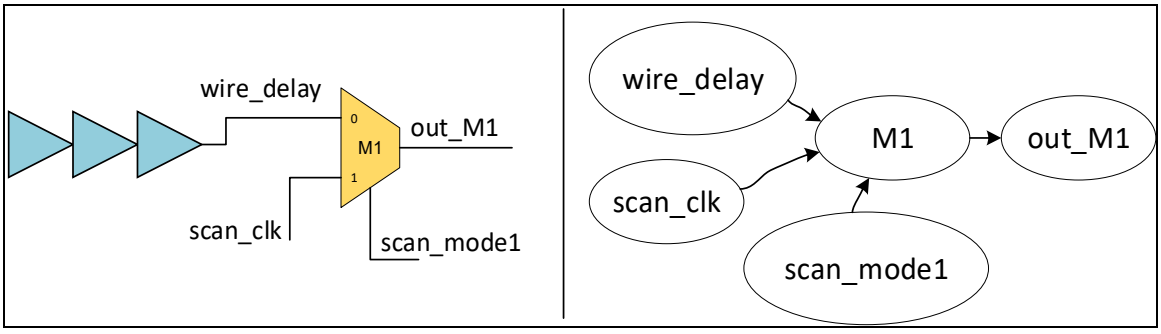

Figure 4.11 Un multiplexeur de test représenté en schéma et en graphe

```
1.  DétecterMultiplexeurs(Graph G, Config conf):
2.       f = CreerFile();
3.       m = CreerFile_de_multiplexeur_de_test();
4.       f.enfiler(conf->départ);
5.       marquer(conf->départ);
6.       Tant que la file est non vide
7.                s = f.defiler();
8.                Pour tout voisin t de s dans G
9.                    Si t non marqué
10.                        Si t est un multiplexeur de test
11.                             m.enfiler(t)
12.                        Fin Si
13.                        f.enfiler(t);
14.                        marquer(t);
15.                    Fin Si
16.               Fin Pour Tout
17.       Fin Tant que
```

Algorithme 4.1 L'algorithme de détection des multiplexeurs de test représenté sous forme de pseudo-code, on note que le principe du parcours en largeur est utilisé ici

4.4.4.3 Analyse des domaines d'horloges

Le but de cette étape est d'attribuer un ou plusieurs domaines d'horloges à chaque registre. En effet, dans certains cas comme dans le mini-MIPS, plusieurs horloges peuvent être capables de contrôler le même registre. C'est le cas de l'unité de calcul et du compteur de programme qui sont des ressources partagées dans le mini-MIPS. Cet algorithme se déroule en deux étapes : le calcul de la distance entre les registres et les multiplexeurs de test et l'analyse des distances pour attribuer des domaines d'horloge aux registres.

Calcul de la distance entre les registres et les multiplexeurs de test

Dans la première fonction *CalculDeDistanceEntreLesRegitresEtLesMuxs* (Algorithme 4.2, ligne 1)**,** chaque distance est enregistrée dans une structure de stockage en deux dimensions. Le nombre de distances calculées correspond au produit entre le nombre de registres et de multiplexeurs de test, soit $988 * 18 = 17784$ distances dans le cas du mini-MIPS.

La deuxième fonction *CalculDistance* mesure la distance dans le graphe entre le sommet de départ et celui d'arrivée. Comme pour la détection des multiplexeurs de test, le calcul de distance est opéré par un algorithme de parcours en largeur. Le parcours du graphe est arrêté si le sommet d'arrivée est trouvé. Lors de la recherche du point d'arrivée, on enregistre dans le sommet courant celui qui le précède dans le tableau *TPrédiction* afin de pouvoir reconstituer le chemin. De la même manière, on garde une trace pour chaque sommet de la distance depuis le point de départ dans le tableau *Distance.* Une fois le chemin reconstitué avec la fonction *LireLeCheminÀRebours,* on compte le nombre de registres parcourus avec la méthode *CompterLesRegistres.* La distance est enfin retournée si le chemin est composé uniquement d'un registre. En effet, si le chemin a parcouru plus d'un registre, cela signifie qu'il a emprunté un chemin de donnée ou traversé un *key unit* dans le cas du mini-MIPS. Or, on souhaite uniquement traverser le chemin d'horloge puisque l'on souhaite se rendre de la sortie d'un multiplexeur à l'entrée d'horloge d'un registre. Dans le cas contraire, une constante qui signifie que le chemin d'horloge n'existe pas est retournée.

```
1.  CalculDeDistanceEntreLesRegitresEtLesMuxs(Graph G,multiplexeurs m, Config conf)
2.       TDistRegMux[N_REGITRES][N_MUX] = InitialiserTableau2Dimensions();

3.       Pour tous les regitres reg dans TDistRegMux
4.          Pour tous les multiplexeurs mux de test dans m
5.                TDistRegMux[reg][mux] = CalculDeDistance(mux,reg,G,conf)
6.          Fin Pour tout
7.        Fin Pour tout
8.
9.
10.
11. CalculDeDistance(Sommet_départ Sd,Sommet_fin Sf,Graph G, Config conf)
12.      Pour tout Sommet s dans Graph
13.          Distance[s] = 0;
14.      Fin Pour tout
15.      TPrédiction = IntialiserTableauDesPrédictions()
16.      f = CreerFile();
17.      f.enfiler(Sd);
18.      marquer(Sd);
19.      Tant que la file f est non vide et que Sf égal non-trouvé
20.               s = f.defiler();
21.               Si s->noms égal Sf->noms
22.                  Quitter boucle;
23.               Fin Si
24.               Pour tout voisin t de s dans G
25.                   Si t non marqué
26.                        Si t->noms égal Sf->noms
27.                             Sf = trouvé;
28.                             Quitter boucle;
29.                        Fin Si
30.                        Distance[t] = Distance[s]+1;
31.                        TPrédection[t] = s;
32.                        f.enfiler(t);
33.                        marquer(t);
34.                    Fin Si
35.                Fin Pour tout
36.       Fin Tant que
37.       Chemin = LireLeCheminÀRebours(Sf,TPrédiction)
38.       nb_registres = CompterLesRegistres(Chemin,Sf)
39.       Si Sf égal trouvé et nb_registres égal 1
40.           Retourner Distance[Sf]
41.       Sinon
42.           Retourner CHEMIN_INVALIDE
43.       Fin Si
```

Algorithme 4.2 L'algorithme qui calcule la distance entre les registres et les multiplexeurs de test représenté sous forme de pseudo-code

Attribution de domaine d'horloge à chaque registre

On peut maintenant attribuer un ou plusieurs domaines d'horloge à chaque registre. Pour ce faire, on compare chaque distance calculée pour un registre. Tel qu'évoqué dans la section précédente, on confond les termes multiplexeurs de test et domaines d'horloges, car la sortie de ces multiplexeurs en est leur source. Le multiplexeur de test situé à la distance minimum du registre est associé en tant que domaine d'horloge du registre. Dans le cas où plusieurs multiplexeurs seraient à la même distance minimum, plusieurs domaines d'horloge sont associés au registre. Dans l'exemple de la Figure 4.12, le registre SREG1 a deux distances calculées, i.e., 7 et 3, le domaine d'horloge du registre SREG1 est donc M1, car c'est le multiplexeur de test le plus près. On compte le point de départ et d'arrivée dans le calcul de distance.

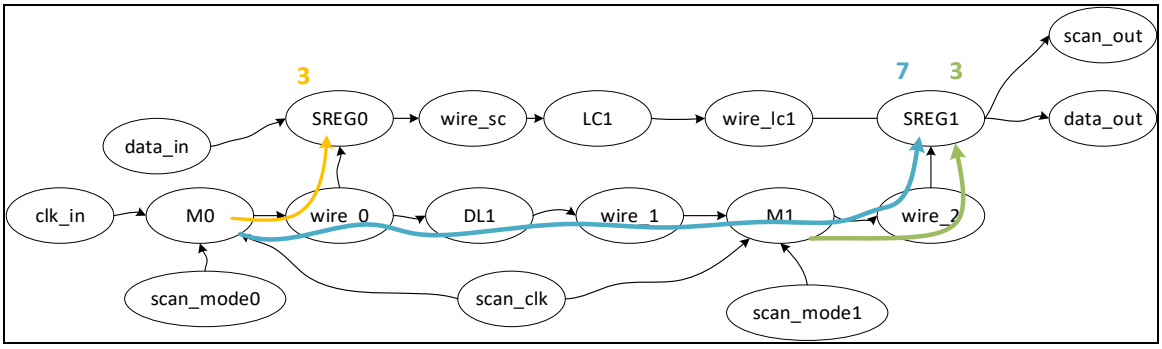

Figure 4.12 Graphe d'un chemin de donnée convergent avec la représentation des distances entre les multiplexeurs de test et les registres

4.4.4.4 Analyse de la dépendance des horloges

L'analyse de la dépendance des horloges est une étape nécessaire pour déterminer les chemins de données convergents. C'est-à-dire les chemins dont le point de départ est un domaine d'horloge et le point d'arrivé est un registre qui peut être accédé par un chemin de donnée et un chemin d'horloge. Cette étape permet donc d'établir quels sont les chemins d'horloge existant dans le circuit. Le résultat sera obtenu sous la forme d'un graphe de multiplexeurs qui permettra de connaître les dépendances entre les domaines d'horloges.

Pour le mini-MIPS et dans l'idéal, notre graphe des dépendances d'horloges devrait ressembler à la dépendance théorique des *key units* vu dans le CHAPITRE 2 (Figure 4.13). En pratique, on verra que du fait des structures complexes dans le chemin d'horloge et le choix de notre algorithme, les dépendances dans notre graphe final seront beaucoup plus nombreuses. L'analyse de la dépendance des horloges regroupe 3 algorithmes qui sont : la contraction de graphe, la définition de la dépendance des horloges à partir du graphe contracté et la recherche de machines à états finis.

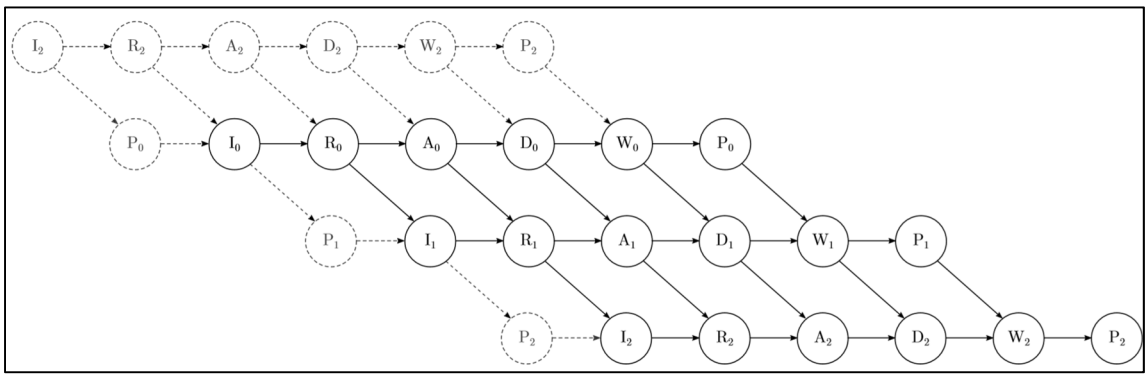

Figure 4.13 Dépendance théorique des horloges du mini-MIPS
Reproduite avec la permission de Fiorentino (2018)

Contraction de graphe (multiplexeurs de test seulement)

L'objectif de cet algorithme est d'obtenir un graphe de multiplexeurs de test qui représente la dépendance entre les domaines d'horloges. Pour y parvenir et à partir du graphe représentant la netlist, l'algorithme va supprimer tous les sommets qui ne sont pas les multiplexeurs de test détectés grâce au programme détaillé à la section 4.4.2.

Le principe de l'algorithme est illustré avec un exemple sur la Figure 4.14. On peut observer que pour chaque sommet supprimé, on connecte ses voisins d'entrée à ses voisins de sortie. Le résultat est un graphe contracté où tous les sommets sont des multiplexeurs de test.

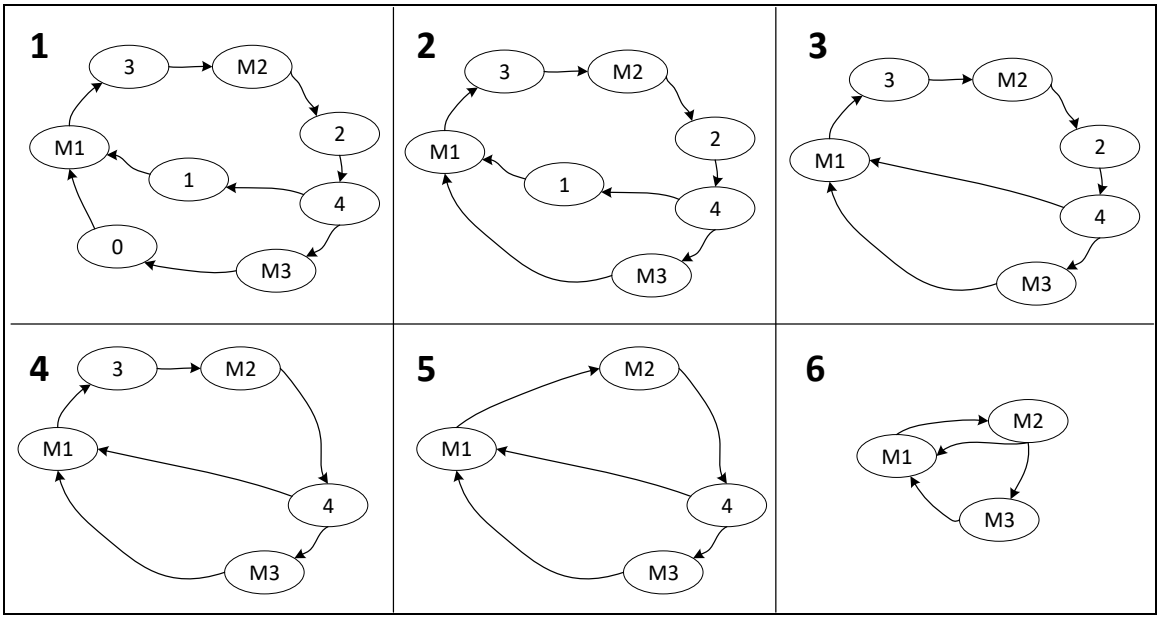

Figure 4.14 Principe de l'algorithme de contraction de graphe, tous les sommets sont supprimés excepté ceux représentant les multiplexeurs de test

La contraction du graphe est réalisée par la fonction *ContractionDeGraphe* décrite en (Algorithme 4.3). Les sommets sont parcourus un à un dans l'ordre de la liste qui représente la netlist. L'algorithme commence par réaliser des vérifications préalables à la suppression du sommet entre les lignes 4 et 10. Ces vérifications sont nécessaires, car, dans certains cas, les sommets doivent être supprimés du graphe. Par exemple lorsqu'ils n'ont pas de voisin d'entrée, il n'y a aucun sommet à connecter au voisin de sortie, le sommet est donc inutile et doit être supprimé. C'est le cas pour les entrées principales du circuit dont la détection est réalisée par *IsInput,* les signaux de contrôle de test par balayage comme *scan_en*, *scan_clock* détecté par *IsScanSignal* ou encore les points de test détectés par *IsTestPoint.* Dans le cas où les sommets ont un retour sur eux-mêmes, les booléens *IsFeedbackOnItslef* et *IsFeedbackOnItselfAndZeroInput* sont nécessaires pour nous permettre de différencier :

- les sommets qui doivent être directement supprimés, car ils n'ont aucun autre voisin d'entrée excepté eux-mêmes;
- les sommets qui doivent être gardés, car ils ont un voisin d'entrée en plus d'eux-mêmes tels que décrits à la Figure 4.15.

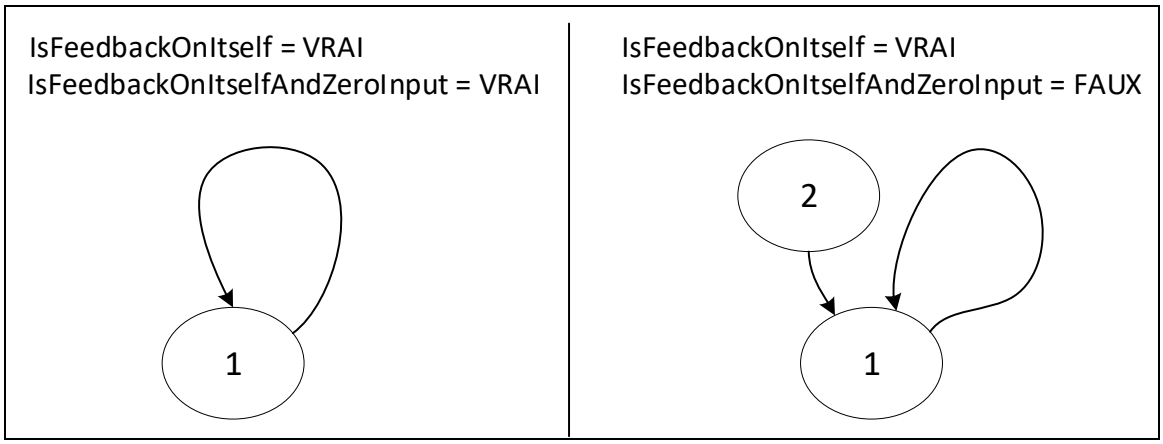

Figure 4.15 Illustration de l'utilisation des booléens IsFeedbackOnItself et IsFeedbackOnItselfAndZeroInput

Si le sommet du graphe correspond au cas présenté à gauche de la Figure 4.15, on peut le supprimer sans connecter ses voisins d'entrée à sa sortie, ce qui reviendrait a ajouté une deuxième boucle de retour sur lui-même. Dans le cas où le sommet correspond au cas de droite de la Figure 4.15, le sommet est géré comme n'importe quel autre élément du graphe.

```
1.  ContractionDeGraphe(config conf, Graphe G)
2.
3.     Pour tout les sommets s dans G
4.         IsMux = DétecterMultiplexeursDeTest(s);
5.         IsFeedbackOnItself = DétecterConnexionSurLuiMême(s);
6.         IsFeedbackOnItselfAndZeroInput = DétecteUniquementUneEntréSurLuiMême(s)
7.         AucuneEntrée = DétectionNombreDEntrée(s);
8.         IsInput = DétecterSiLeSommetEstUneEntrée(s);
9.         IsScanSignal = DétecterSiLeSommetEstUnSignalScan(s);
10.        IsTestPoint = DétecterSiLeSommetEstUnSignalTestPoint(s);
11.
12.        Si IsTestPoint OU IsScanSignal OU IsInput OU AucuneEntrée OU IsFeedbackOn
    ItselfAndZeroInput
13.            Pour tout les voisins v connecté en sortie du sommet s
14.               SupprimerArrete(s,v,G);
15.            Fin Pour tout
16.            SupprimerSommet(s,G)
17.        Sinon Si IsMux égal Faux
18.            Pour tous les voisins ve en entrée du sommet s
19.               Si IsFeedbackOnItself égal Vrai
20.                  Si s contient ve en entrée
21.                     Continuer;
22.                  Fin si
23.               Fin Si
24.               Pour tous les voisins vs en sortie du sommet s-
25.                  Si IsFeedbackOnItself égal Vrai
26.                     Si s contient vs en sortie
27.                        Continuer;
28.                     Fin si
29.                  Fin Si
30.                  SupprimerArrete(ve,s,G);
31.                  SupprimerArrete(s,vs,G);
32.                  AjouterArrete(ve,vs,G);
33.               Fin Pour tous
34.            Fin Pour tous
35.            SupprimerSommet(s,G);
36.        Fin Si
37.    Fin Pour tout
```

Algorithme 4.3 L'Algorithme de contraction de graphe représenté sous forme de pseudo-code

Définition de la dépendance des horloges à partir du graphe contracté

Une fois le graphe contracté obtenu, il ne reste plus qu'à extraire les relations entre les horloges. Par exemple, sur le graphe de la Figure 4.16 (côté gauche), on sait que pour générer son horloge, M1 est dépendant des horloges générées par M2 et M3. Il faut donc maintenant analyser le graphe contracté pour stocker les informations de dépendance dans une structure de stockage à deux dimensions comme dans la Figure 4.16 (côté droit).

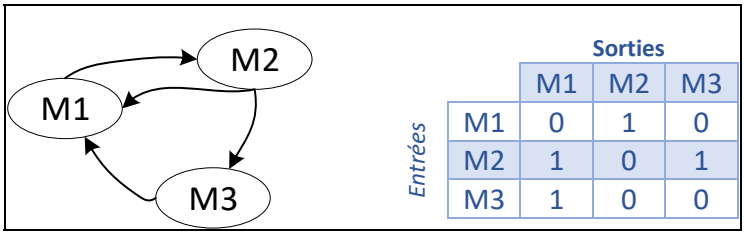

Figure 4.16 La dépendance des horloges représentée sous forme de graphe et de matrice

Recherche des machines à états finis

La recherche de machines à états finis intervient juste après la contraction du graphe. Le principe de cette recherche part de l'hypothèse suivante :

- un domaine d'horloge comporte une machine à états finis lorsqu'un sommet représentant un domaine d'horloge boucle sur lui-même.

Cette hypothèse est suffisante pour un design de circuit comme la machine à états finis ITC99 du banc d'essai. Mais on verra dans le chapitre sur les résultats que cette règle ne fonctionne pas pour la détection des FSM d'un circuit complexe comme le mini-MIPS. On verra dans le CHAPITRE 6 sur les résultats que les quelques détections réalisées sur le mini-MIPS sont le fruit d'erreurs de l'algorithme.

4.4.4.5 Recherche des chemins de données convergents

Le rôle de la recherche de chemins de données convergents est de pouvoir générer un graphe de multiplexeurs de test comme dans l'analyse de la dépendance des horloges. Cependant, à la différence de l'étape précédente, les domaines d'horloges seront uniquement connectés entre eux si : un chemin de donnée est détecté entre deux registres dont leurs domaines d'horloge sont connectés. Ce résultat est obtenu par deux algorithmes : une contraction de graphe et l'algorithme de recherche des chemins de données convergents.

Contraction de graphe (multiplexeurs de test et registres)

De la même manière que dans l'analyse de la dépendance des horloges, on réalise une contraction à partir du graphe représentant la netlist initialement construite par le programme. Contrairement à la précédente contraction, on gardera deux types de sommets : les multiplexeurs de test et les registres. Mise à part cette différence, l'algorithme utilisé est le même que dans la section 4.4.4, on ne le redétaillera donc pas ici. Le type de graphe obtenu est illustré à la Figure 4.17. Dans cet exemple, on voit très clairement que le circuit peut contenir des registres dans les chemins d'horloges illustrés par les sommets REGCLK. On ne pourrait donc pas utiliser seulement ce graphe pour trouver les chemins de données convergents, car il serait difficile de différencier les chemins d'horloge des chemins de données. En effet, les deux types de chemins comportent des registres.

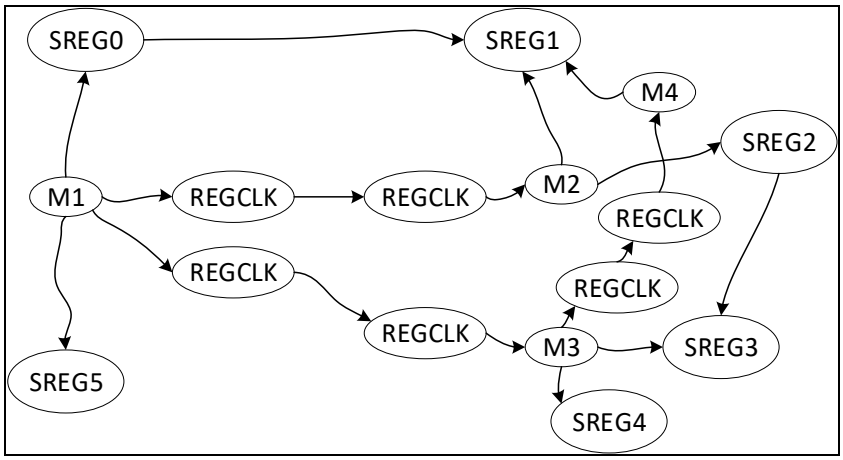

Figure 4.17 Graphe représentant le résultat de la contraction du graphe initial en gardant les multiplexeurs de test et les registres

Algorithme de recherche de chemins convergents

Dans cet algorithme, détaillé en Algorithme 4.4, on utilise tous les résultats calculés par les précédentes fonctions pour trouver les chemins de données convergents. C'est la dernière étape avant de pouvoir déterminer le nombre et la connexion des signaux *scan_modes* par coloration de graphe.

Pour toutes les combinaisons de registres et multiplexeurs de test, l'algorithme vérifie si le registre est un point de convergence. Pour rappel, on définit un point de convergence comme un endroit du circuit où, à partir d'un domaine horloge, les chemins d'horloge et de donnée convergent sur un même registre. Si le registre est un point de convergence, on connecte ensemble dans un graphe, le multiplexeur de test de départ et le multiplexeur de test connecté au point de convergence. Par exemple pour trouver le chemin de données convergent formé par le chemin de donnée M1-SREG0-SREG1 et le chemin d'horloge M1-REGCLK-REGCLK-M2-SREG1 représenté à la Figure 4.17. Le programme procède aux étapes suivantes :

1. À la ligne 7, on vérifie grâce à l'analyse de la dépendance des horloges si le registre SREG1 est bien au moins contrôlé par un multiplexeur de test dépendant de M1. Cette

vérification initiale permet de s'assurer que certains domaines d'horloges des registres SREG0 et SREG1 sont bien connectés.

2. Si M2 est dépendant de M1 alors le programme essaie de trouver un chemin de donnée entre M1 et SREG1 (ligne 8). Pour ce faire, on utilise un algorithme de parcours en profondeur chargé de trouver des chemins entre M1 et SREG1.
   a. Pour que le chemin soit considéré comme valide, il faut qu'il respecte deux critères :
      - le nombre de sommets traversé doit être égal à 3,
      - excepté le point de départ, le chemin est uniquement composé de registres.
3. L'algorithme considère donc le chemin formé par M1-SREG0-SREG1 comme valide.
4. À la ligne 9, pour tous les multiplexeurs de test connectés à SREG1, on peut donc maintenant vérifier précisément quel multiplexeur de test est connecté avec M1. En l'occurrence dans notre exemple on sait que M1 est connecté à M2.
5. On peut donc ajouter une arête entre les sommets M1 et M2 dans notre graphe final, car on sait que ces deux domaines d'horloges sont reliés par un chemin de donnée.

```
1.  TrouverLesCheminsConvergents(Graph G, Multiplexeur_de_test MuxList)
2.
3.  GraphMuxDesCheminsConvergents = InsérerTousLesMux(MuxList);
4.
5.  Pour tout les multipexeurs de test mux dans MuxList
6.    Pour tout les registres reg dans G
7.        Si domaine_dhorloge(reg) est dépendant de mux
8.            Si chemin de donnée existe entre mux et reg
9.                Pour tout les multipexeurs de test mux2 connecté à reg
10.                  Si mux2 est dépendant de mux et qu'il ne sont pas déja connecté
11.                        AjouterUneArrête(mux,mux2,GraphMuxDesCheminsConvergents)
12.                  Fin Si
13.               Fin Pour tout
14.           Fin Si
15.       Fin Si
16.   Fin Pour tout
17. Fin Pour tout
```

Algorithme 4.4 L'algorithme de recherche des chemins de données convergents représenté sous forme de pseudo-code

4.4.4.6 Coloration de graphe pour déterminer le nombre et la connexion des signaux scan_modes

L'assignation des signaux de contrôle des multiplexeurs de test *scan_modes* est réalisée par trois fonctions décrites sur l'Algorithme 4.5 : *AssignerScanMode*, *ColorationDeGraph_P* et *ColorationDeGraph.*

Le rôle de la fonction *AssignerScanMode* est de procéder à la coloration du graphe généré par la recherche des chemins de données convergents. L'objectif est double : minimiser le nombre de signaux *scan_modes* dans le circuit tout en permettant de couvrir tous les chemins de données convergents par des tests *launch-on-capture*. Comme on a pu le voir dans les CHAPITRE 1 et CHAPITRE 3, il faut que les domaines d'horloges successifs dans un chemin de données convergent soient contrôlés par des signaux *scan_mode* différents. Par exemple sur la Figure 4.18, pour que l'impulsion de capture en vert soit générée, le multiplexeur M0 doit sélectionner l'entrée 1 tandis que M1 doit sélectionner l'entrée 0. Cela implique donc forcément des signaux de contrôle différent.

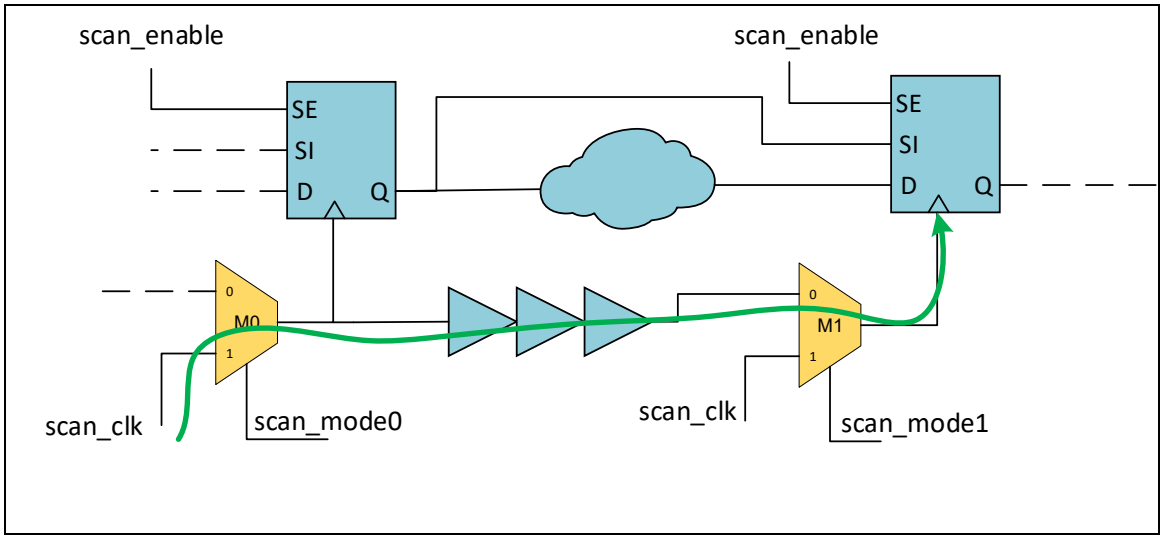

Figure 4.18 Schéma qui représente un chemin convergent du mini-MIPS, on illustre en vert le chemin de l'impulsion sur le signal scan_clk vers le registre qui constitue le signal de capture

Le but original de l'algorithme de coloration de graphe est d'associer une couleur pour chaque sommet d'un graphe G de manière à ce que deux sommets adjacents ne soient pas de la même couleur (Figure 4.19). Bien sûr dans notre cas, les couleurs correspondent aux signaux de contrôle *scan_mode*. Le processus est effectué par les deux fonctions *ColorationDeGraph_P* et *ColorationDeGraph*. Les fonctions font appel à un algorithme standard de *backtracking*. Pour chaque sommet à colorer, on vérifie si aucun des voisins ne possède pas déjà le même signal *scan_mode* avec *IsSafeToColor*, si oui on assigne le signal au sommet, sinon, l'algorithme tentera de nouveau d'assigner un signal avec un signal *scan_mode* différent. Le processus s'arrête quand l'algorithme a parcouru tous les sommets du graphe.

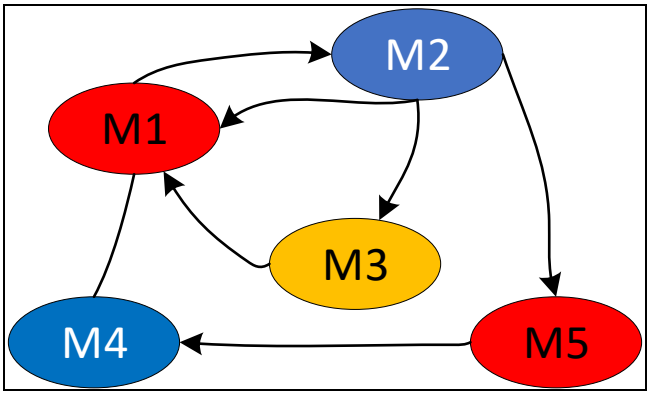

Figure 4.19 Exemple de coloration d'un graphe de multiplexeur de test représentant les domaines d'horloge, on associe à chaque couleur un signal de contrôle scan_mode

Si l'algorithme n'arrive pas à trouver une solution, la fonction *AssignerScanMode* le relance en incrémentant le nombre de signaux *scan_modes*. Le programme répète la procédure jusqu'à trouver le nombre de signaux qui résout le problème initial.

Deux autres types de signaux *scan_mode* s'ajoutent à ceux déterminés par l'algorithme de coloration de graphe :

Le 1er type correspond aux signaux *scan_mode* qui contrôlent des multiplexeurs de test sans voisin de sortie dans le graphe de recherche des chemins de données convergents. En effet, certains domaines d'horloges peuvent être reliés dans le graphe d'analyse de la dépendance des horloges, mais non connectés dans le graphe de recherche des chemins de données convergents. Cela est dû au fait qu'aucun chemin de donnée convergent n'a été trouvé entre ceux deux domaines d'horloge. Cela signifie donc qu'il n'y a aucun chemin à tester à partir de ce domaine d'horloge. On ne devra donc jamais propager une impulsion de lancement à travers les multiplexeurs de test qui correspondent à ce type de domaine d'horloge. Pourtant, certains de ces domaines d'horloges peuvent être colorés de la même couleur que d'autres domaines d'horloge qui eux doivent propager une impulsion de lancement. On doit donc séparer le contrôle de ces deux types de domaines d'horloge en créant un nouveau signal de type *scan_mode*. Ce type de signal est labélisé « *end_graph* » car ces domaines d'horloges correspondent aux sommets qui n'ont pas de voisin de sortie dans le graphe de la recherche des chemins de données convergents.

Le 2ème type correspond au cas des domaines d'horloge contrôlant des machines à états finis. En effet, le signal *scan_mode* qui contrôle le multiplexeur de test qui correspond à ce type de domaine d'horloge sera utilisé pour deux types de tests. Le premier test correspond au *launch-on-capture* pour tester les chemins convergents avec la méthode des multiplexeurs présentés en revue de littérature et dans le CHAPITRE 3. Le deuxième type est le test de machine à états finis présenté au CHAPITRE 3. Or, nous verrons dans le CHAPITRE 5 que les tests de machine à états finis sont séparés des tests des chemins convergents. On doit alors pouvoir contrôler les multiplexeurs de test (qui seront remplacés par les modules *FSM_test* dans la 3ème phase) qui contrôlent des machines à états finis de manière individuelle. On doit donc créer un nouveau signal *scan_mode* pour chaque multiplexeur de test qui contrôle une machine à états finis. Ces signaux seront utilisés à la fois de manière individuelle lors des tests de machine à états finis et à la fois de manière groupée avec l'autre signal *scan_mode*

(déterminé par la fonction de coloration de graphe) lors de l'emploi de la méthode des multiplexeurs de test. Le programme indiquera à l'utilisateur à la fin de son exécution quels signaux *scan_mode* peuvent être utilisés en même temps.

```
1.  AssignerScanMode(Graph G, List HorlogesQuiContrôlentDesFSM)
2.      SolutionExiste = FAUX;
3.      Nombre_de_scan_mode = 2;
4.      Tant que SolutionExiste égal FAUX
5.          SolutionExiste = Coloration(G,Nombre_de_scan_mode);
6.          Nombre_de_scan_mode = Nombre_de_scan_mode + 1;
7.      Fin Tant que
8.
9.      Pour tous les sommet s dans G
10.         Si s n'a pas de sommet connecté en sortie
11.             s.assigner_scan_mode(end_graph)
12.         Fin Si
13.         Si HorlogesQuiContrôlentDesFSM contient s
14.             Si s.scan_mode est utilisé par un autre sommet s de G
15.                 s.assigner_scan_mode(NouveauScanMode());
16.             Fin Si
17.         Fin Si
18.     Fin Pour tous
19.
20.
21. ColorationDeGraph_P(Graph G, Nb_signaux_scan_modes)
22.     list_scan_mode = Créer_liste_scan_mode(Nb_signaux_scan_modes);
23.     sommet_courant = G.début;
24.     Si ColorationDeGraph(sommet_courant,G,list_scan_mode) égal FAUX
25.         Retourner FAUX;
26.     Fin Si
27.     Retourner VRAI;
28.
29.
30. ColorationDeGraph(Sommet sommet_courant, Graph G,List list_scan_mode)
31.
32.     Si sommet_courant égal fin_du_graph
33.         retourner VRAI;
34.     Fin Si
35.
36.     Pour tous les scan_modes scm dans list_scan_mode
37.         Si IsSafeToColor(sommet_courant,scm)
38.             sommet_courant.assigner_scan_mode(scm)
39.             Si ColorationDeGraph(sommet_courant+1,G,list_scan_mode)
40.                 Retourner VRAI;
41.             sommet_courant.supprimer_assignement_scan_mode()
42.             Fin Si
43.         Fin Si
44.     Fin Pour tous
45.
46.     retourner FAUX;
```

Algorithme 4.5 L'algorithme de coloration de graphe qui permet de déterminer le nombre et la connexion de signaux de contrôle de type scan_mode représenté sous forme de pseudo-code

4.4.4.7 Préparation de la connexion des modules de contournement *Bypass_KU*

Tel que nous l'avons vu dans le CHAPITRE 2 et CHAPITRE 3, la structure du réseau d'horloge du mini-MIPS est composée de *key units*. Ces structures complexes à contrôler bloquent le passage de l'impulsion de capture. En effet, dans la topologie de design original de la méthode d'Octasic décrite en revue de littérature et illustrée du côté droit de la Figure 4.20 , ces structures ne sont pas présentes. Comme expliqué dans le CHAPITRE 3, on doit donc contourner les *key units* pour que l'impulsion se rende au registre de capture.

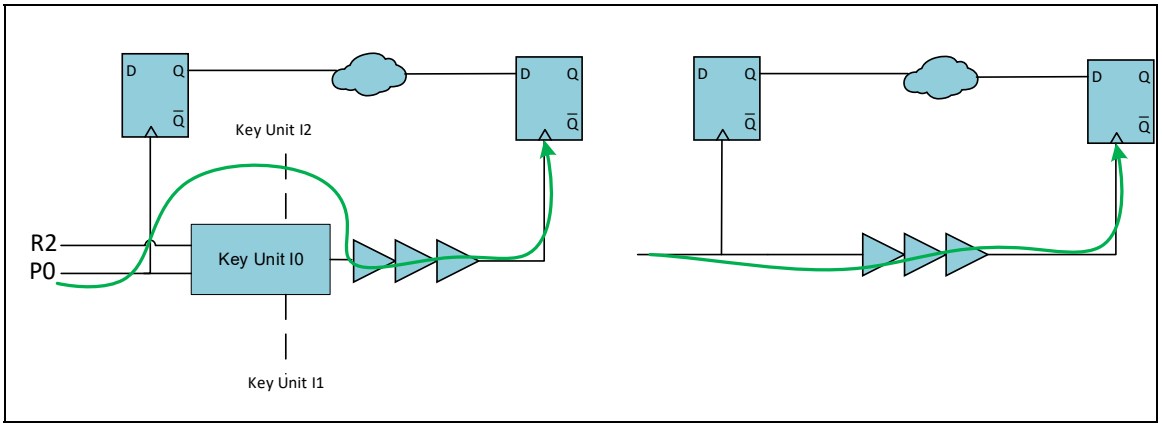

Figure 4.20 Comparaison d'un chemin de donnée convergent dans le mini-MIPS (à gauche) et selon la topologie de design d'Octasic (à droite), on y représente en vert le chemin que doit emprunter l'impulsion de capture

Ces modules de contournement illustré en Figure 4.21 sont placés en aval des *key units*. Ils disposent de 11 entrées d'horloges différentes pour satisfaire toutes les combinaisons de chemin d'horloge possible. Le rôle de cette étape du programme est donc de renseigner quelles horloges doivent être connectées aux différents *Bypass_KU* du mini-MIPS.

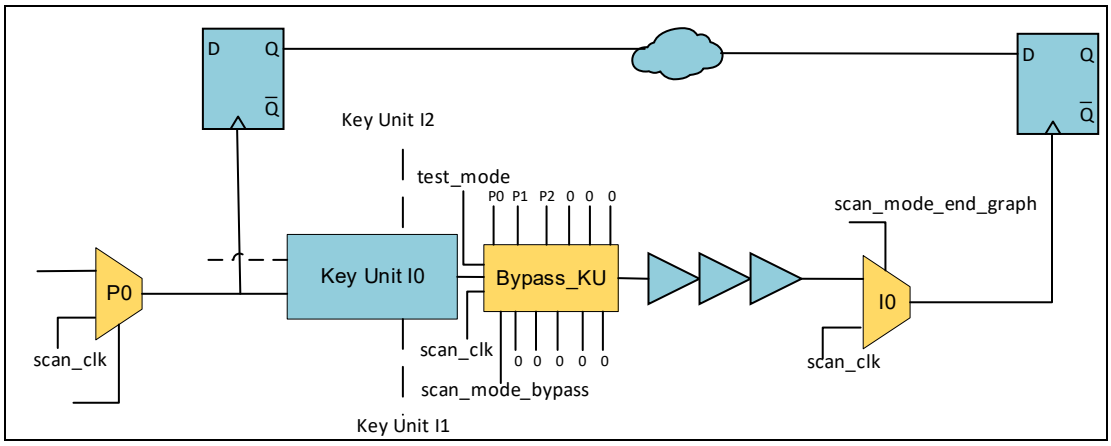

Figure 4.21 Schéma d'un chemin convergent dans le mini-MIPS avec ses structures de test

Pour ce faire, le programme s'aide du graphe généré par la recherche des chemins convergents. Par exemple, pour obtenir les connexions de la Figure 4.21, il faut qu'une partie du graphe généré dans l'étape d'analyse de la dépendance des horloges corresponde à la Figure 4.22.

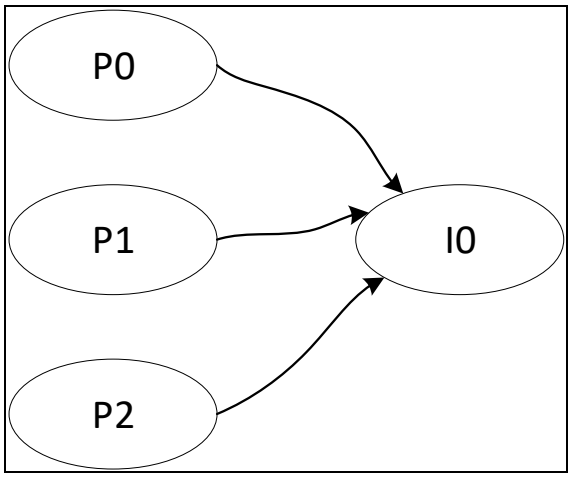

Figure 4.22 Graphe représentant la dépendance de l'horloge I0 aux horloges P0,P1 et P2. Il est utilisé pour la connexion des signaux d'horloge de *Bypass_KU*

4.5 Insertion des structures de test spécialisées

L'insertion des structures de test spécialisées est la dernière phase qui apporte des modifications au circuit. Dans cette partie, on exécute deux scripts à l'aide de Tessent. Le premier est chargé d'insérer les modules de test de machine à états finis *FSM_test*. Le deuxième est spécifique au mini-MIPS et permet d'insérer les modules de contournement *Bypass_KU*. Les deux scripts s'aident des fichiers générés par le programme d'analyse et connexions des signaux de contrôle pour savoir où insérer les modules. Ils ne font donc qu'interpréter les instructions générées par la 2ème phase. Pour cette raison, les scripts ne seront pas plus détaillés. Ils sont néanmoins disponibles en ANNEXE II et ANNEXE III. Enfin, cette 3ème phase se termine par un passage dans notre outil de synthèse qui nous permet d'aplatir la netlist et d'extraire les délais qui seront nécessaires pour la simulation externe des vecteurs de test. Le script chargé de ce processus est disponible en ANNEXE IV.

4.6 Conclusion

Dans ce chapitre, nous avons vu de quelle manière les structures spécifiques présentées dans le CHAPITRE 3 sont insérées et connectées dans les circuits à l'aide du processus de testabilité automatisée que nous venons juste de présenter. La méthode proposée s'aide des solutions commerciales déjà existantes avec l'utilisation de Tessent et Genus pour réaliser toutes les actions mécaniques. Les étapes d'analyse et de décision sont, elles, toutes concentrées sur le programme d'analyse et connexion des signaux de contrôle du test développé pour l'occasion. Bien que ce flot de testabilité ait été développé dans le but d'être le plus automatisé possible, il reste tout de même du travail au concepteur du circuit. Ainsi, certaines tâches, comme l'identification des horloges principale et secondaire du circuit restent à la charge du concepteur. Il devra aussi veiller à remplir le fichier de configuration nécessaire au bon fonctionnement de notre programme. De plus, chaque circuit est différent et peut réserver son lot de surprises, le concepteur doit donc savoir s'adapter à la particularité du design. Pour nous, ce sont les structures de *key units* qui nous ont posé problème, car elles

bloquent le chemin de nos impulsions de capture. Nous avons donc dû mettre en place des structures spécialisées pour les contourner. Enfin, le concepteur doit aussi s'assurer de remplir les fichiers de procédure nécessaire à la génération automatique des vecteurs de test. C'est ce que nous allons voir dans le prochain chapitre.

CHAPITRE 5

GÉNÉRATION AUTOMATIQUE DE VECTEURS DE TEST

5.1 Introduction

L'objectif de ce chapitre est de présenter l'ensemble des vecteurs de test générés à l'aide de l'ATPG sur tous les circuits étudiés. La première partie nous informe succinctement sur les étapes de la phase de génération des vecteurs de test. La deuxième partie sur le plan d'expérimentation décrit l'ensemble des expériences réalisées sur l'ensemble de circuits du banc d'essai présenté au CHAPITRE 2 pour générer les vecteurs. De plus, elle détaille l'état des signaux de contrôle *scan_modes* connectés à l'ensemble des multiplexeurs de test. Les résultats de ces expérimentations seront analysés et discutés dans le prochain CHAPITRE 6 sur les résultats. En prenant comme exemple le mini-MIPS, la dernière partie détaille l'ensemble des procédures constituant les tests. C'est cette description des tests qui permet à l'ATPG de générer les vecteurs de test. Elle est accompagnée d'exemples de fichiers de procédure qui sont nécessaires pour que Tessent, notre outil de testabilité, puisse simuler les tests.

5.2 Présentation du flot de générations des vecteurs de test des circuits

La phase de génération automatique des vecteurs de test se déroule en 2 étapes (Figure 5.1):

- la génération automatique des vecteurs de test qui donne un taux de couverture de panne résultant d'une simulation logique sans délai par l'ATPG;
- la simulation des vecteurs générés par l'ATPG réalisée avec un outil externe de simulation, dans notre cas Modelsim.

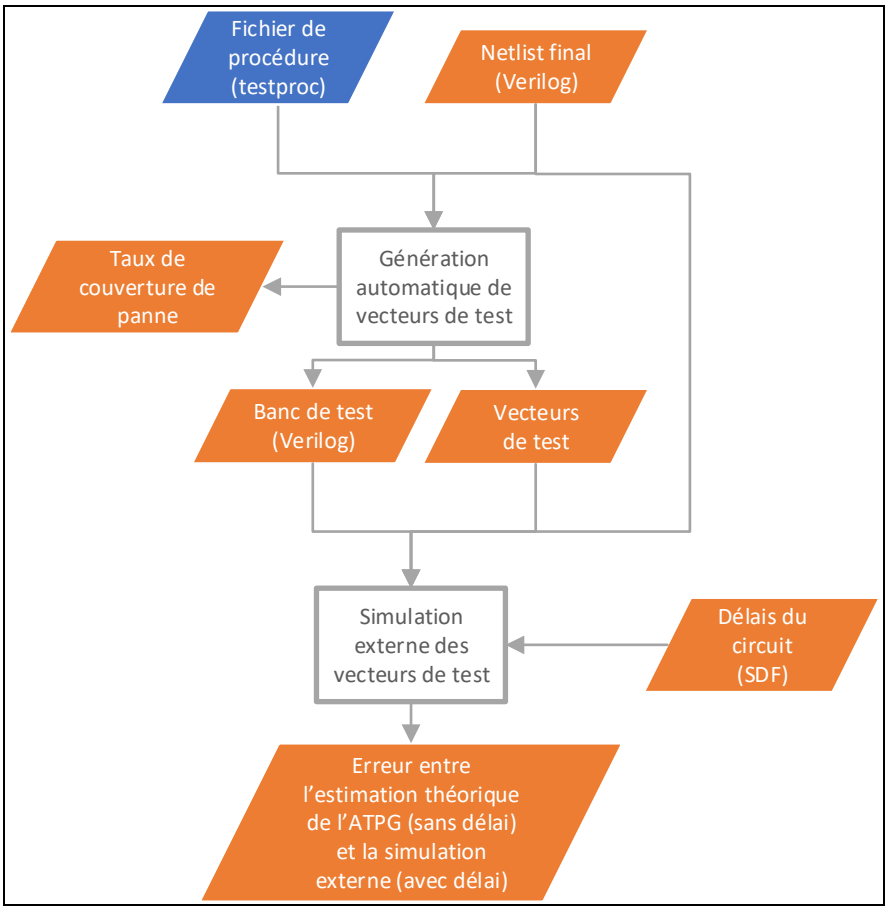

Figure 5.1 Déroulement de la phase de génération automatique des vecteurs de test des circuits

Contrairement à la simulation de l'ATPG, la simulation externe des vecteurs de tests prend en compte les délais du circuit. Sa seule fonction est de vérifier que les vecteurs de test générés par l'ATPG sont bien exécutables par le vrai circuit avec ses délais.

Ces deux étapes sont réalisées avec un script disponible en ANNEXE V exécuté par notre logiciel de testabilité, Tessent. Ce script permet de lancer la génération automatique des vecteurs de test et de créer automatiquement un banc de test en verilog simulable par un logiciel externe comme Modelsim. Ce banc de test nous permet de simuler les vecteurs de test généré par la 1re étape. Les procédures de tests exécutées par le script sont définies dans un fichier de procédure (.testproc) qui est expliqué en partie dans la section 5.4 de ce chapitre.

5.3 Plan d'expérimentation

On décrit dans cette partie l'ensemble des générations de vecteurs de test réalisées sur les trois types de circuits présentés dans le CHAPITRE 2 : les machines à états finis issues du jeu de circuits ITC99, le pipeline asynchrone typique de l'architecture d'Octasic et le mini-MIPS asynchrone.

Chaque section présentera les procédures de test sous forme de tableau lorsque cela est possible. Ces tableaux présentent l'état des signaux de type *scan_mode* utilisé pour contrôler les multiplexeurs de test ou les modules de test de machine à états finis *FSM_test* insérés dans les circuits. La lettre **L** sera attribuée à un signal *scan_mode* lorsque celui-ci est utilisé en mode « launch ». C'est-à-dire que les multiplexeurs de test connectés au signal sont utilisés pour propager l'impulsion de lancement. À l'inverse, la lettre **C** est utilisée pour signifier que les multiplexeurs reliés au signal *scan_mode* sont utilisés en mode de « capture ». C'est-à-dire que les multiplexeurs associés à ce signal *scan_mode* sont utilisés pour capturer la réponse du chemin de donnée du test de transition.

5.3.1 Machines à états finis issues de ITC99

On rappelle que les circuits issus du benchmark ITC99 sont au nombre de 3, leurs caractéristiques ont été détaillées dans le CHAPITRE 2 . Le processus de testabilité pour ces trois circuits est différent des deux autres types de circuits du banc d'essai. En effet, à la différence du pipeline typique des circuits d'Octasic et du mini-MIPS, ces circuits possèdent une horloge unique. Il n'y a donc pas besoin du programme de connexion des signaux de contrôle, car les circuits possèdent un unique signal *scan_mode*. Pour ces circuits, le processus de testabilité consiste donc à insérer la chaine de balayage et le module de test de machine à états finis. L'intérêt du test de ces circuits réside dans le fait qu'il est possible de comparer nos résultats de taux de couverture de panne avec des références extérieurs à ce mémoire. Ce processus nous permettra donc de valider le bon fonctionnement du module de

test machine à états finis qui était inexistants dans la méthode de test original présenté en revue de littérature. La génération des vecteurs pour ces circuits mettra en exécution une simple procédure de *launch-on-capture* présentée dans le CHAPITRE 3. Au cours de ce test, les mêmes registres seront à la fois à l'origine du lancement et de la capture du test de transition (Tableau 5.1).

Tableau 5.1 État du signal de contrôle scan_mode
lors de la procédure de *launch-on-capture*

Sous-procédure	**Signaux de contrôle des multiplexeurs de test**
	scan_mode
LOC FSM	LC

5.3.2 Pipeline typique des circuits d'Octasic

Le pipeline typique des circuits d'Octasic nous permet de vérifier le fonctionnement de notre flot de testabilité, plus particulièrement des phases 1 et 2, avant de le soumettre à un circuit beaucoup plus complexe comme le mini-MIPS. La phase 3 du processus de testabilité lui est inutile puisqu'aucune structure spécialisée comme le module de contournement *Bypass_KU* ou le module de test machine à états finis *FSM_test* ne lui est nécessaire. Pour ce circuit, 2 types de tests sont réalisés : un test collé-à et un test de transition *launch-on-capture* à vitesse nominale.

5.3.2.1 Test collé-à

Le test collé-à nous permet d'atteindre le taux maximum de couverture de panne réalisable par la disposition de structure de test inséré dans le circuit. Il nous permet de déterminer une valeur maximum de taux de couverture de panne lors de futurs tests de *launch-on-capture*. En effet, le test collé-à peut couvrir certain cas que le test de transition ne peut pas. Par

exemple, le test des broches de réinitialisation des registres ou encore de la logique combinatoire entre un registre et une sortie du circuit.

5.3.2.2 *Launch-on-capture* à vitesse nominale (at-speed)

La procédure de test de transition est découpée en deux sous procédures (Tableau 5.2). Comme le circuit possède deux signaux de type *scan_mode*, le rôle de lancement et de capture des multiplexeurs de test est inversé dans chaque sous-procédure tel que présenté en revue de littérature et au CHAPITRE 3.

Tableau 5.2 État des signaux de contrôle *scan_mode* lors de la procédure de test de *launch-on-capture* entièrement à vitesse nominale du pipeline typique des circuits d'Octasic

Sous-procédure	**Signaux de contrôle des multiplexeurs de test**	
	scan_mode_0	scan_mode_1
Launch scan_mode_0	L	C
Launch scan_mode_1	C	L

5.3.3 Mini- MIPS

Le mini-MIPS est le circuit le plus complexe de l'ensemble de circuits du banc d'essai. Il regroupe à la fois des topologies du pipeline typique d'Octasic et des machines d'état finis issues de ITC99 en plus de plusieurs autres types de chemins de donnée tels que présentés dans le CHAPITRE 3. Le circuit nous permet donc d'exploiter l'ensemble du processus de testabilité introduit dans le CHAPITRE 4. L'objectif des tests suivants est d'atteindre le plus haut taux de couverture de panne possible avec la disposition des structures de test proposés

dans le CHAPITRE 3. Pour ce faire, trois tests pour la génération de vecteur sont réalisés : le test collé-à, un test de transition *launch-on-capture* entièrement à vitesse nominale et un test de transition de *launch-on-capture* partiellement à vitesse nominale puisqu'il intègre une étape de *launch-on-capture* synchrone.

5.3.3.1 Test collé-à

Comme dans le pipeline typique des circuits d'Octasic, ce test nous permet de déterminer le taux maximum de couverture de panne possible avec les structures de test insérées dans le circuit.

5.3.3.2 *Launch-on-capture* entièrement à vitesse nominale (at-speed)

La procédure de *launch-on-capture* entièrement à vitesse nominale peut être décomposée en deux parties qui utilisent au final 7 sous-procédures (Tableau 5.3). La première partie pernet de couvrir les pannes des chemins de données convergents. De plus, on a vu dans les CHAPITRES 3 et 4 que ces procédures couvriront aussi les chemins indirectement convergents de catégorie 1 (ICC1D). Ces procédures sont composées de quatre *launch-on-capture* et utilisent le principe des multiplexeurs de test vu en revue de littérature et dans le CHAPITRE 3. Dans chacun de ces tests, uniquement un signal *scan_mode* est activé en mode de lancement tandis que les autres sont en mode capture.

La deuxième partie de la procédure est composée de 3 autres *launch-on-capture* chargé de tester les machines à états finis. Ces *launch-on-capture* ne peuvent pas être réalisés parallèlement, car les horloges concernées (A0, A1, A2) sont directement reliées entre elles et peuvent former une boucle combinatoire si tous les modules *FSM_test* sont activés en mode de capture en même temps. La technique employée ici est celle décrite dans le CHAPITRE 3 et est la même que pour les circuits issus du benchmark ITC99. On en reparlera aussi dans la section 5.4 de ce chapitre qui concerne la définition des procédures de test.

Tableau 5.3 État des signaux de contrôle *scan_mode* lors de procédure de *launch-on-capture* entièrement à vitesse nominale pour le mini-MIPS

Sous-procédure	Signaux de contrôle des multiplexeurs de test					
	Scan_mode_0	Scan_mode_1	Scan_mode_1A	Scan_mode_2	Scan_mode3	End_graph
Launch scan_mode_0	L	C	C	C	C	C
Launch scan_mode_1	C	L	L	C	C	C
Launch scan_mode_2	C	C	C	L	C	C
Launch_scan_mode_3	C	C	C	C	L	C
LOC FSM A0	L	L	LC	L	L	C
LOC FSM A1	L	L	L	LC	L	C
LOC FSM A2	L	L	L	L	LC	C

5.3.3.3 *Launch-on-capture* partiellement à vitesse nominale

Cette procédure reprend les mêmes étapes que la précédente, la seule différence est qu'on intègre ici un test de *launch-on-capture* synchrone (Tableau 5.4). Ce test nous permettra d'augmenter encore notre taux de couverture de panne et sera utile lors de l'analyse des pannes restantes. On en reparlera plus en détail dans la section 5.4 de ce chapitre.

Tableau 5.4 État des signaux de contrôle *scan_mode* lors de la procédure de *launch-on-capture* partiellement à vitesse nominale pour le mini-MIPS

Sous-procédure	Signaux de contrôle des multiplexeurs de test					
	Scan_mode_0	Scan_mode_1	Scan_mode_1A	Scan_mode_2	Scan_mode3	End_graph
Launch scan_mode_0	L	C	C	C	C	C
Launch scan_mode_1	C	L	C	C	C	C
Launch scan_mode_2	C	C	C	L	C	C
Launch_scan_mode_3	C	C	C	C	L	C
LOC FSM A0	L	L	LC	L	L	C
LOC FSM A1	L	L	L	LC	L	C
LOC FSM A2	L	L	L	L	LC	C
LOC synchrone	LC	LC	LC	LC	LC	LC

5.4 Définition des procédures de test

La section suivante décrit la définition des fichiers de procédures de test. Ces fichiers sont un préalable nécessaire à la génération des vecteurs de test par l'ATPG et doivent être rédigés par l'ingénieur responsable des tests. Seulement quelques extraits les plus pertinents du fichier sont présentés dans cette section. On peut retrouver l'ensemble du script en ANNEXE VI.

Le mini-MIPS est ici pris comme exemple, car c'est le circuit le plus complexe sur lequel nous avons travaillé. Il regroupe donc tous les cas de tests que l'on peut trouver sur le pipeline typique d'Octasic et des machines à états finis issus du jeu de circuit ITC99.

5.4.1 Définition des horloges

Dans cette partie de la procédure, on vient déclarer toutes les horloges dont l'ATPG aura besoin pour exécuter les différents tests de transition *launch-on-capture*. On a vu dans la section précédente que le mini-MIPS avait 4 procédures de *launch-on-capture* pour tester les

chemins de données convergents. Cela équivaut à une procédure par groupe principal de signaux *scan_mode* en mode de lancement (scan_mode0, scan_mode1 et scan_mode1A, scan_mode2, scan_mode3). On exclut le signal end_graph, car il est toujours en mode de capture. Pour chaque procédure, on doit définir toutes les horloges utiles à la procédure et exclure de la déclaration les horloges inutilisées.

On peut prendre comme exemple la procédure de *launch-on-capture* qui utilise scan_mode2 en mode de lancement. La répartition des horloges est illustrée à la Figure 5.2 et la déclaration de ces horloges est présentée en Figure 5.3. Ce classement des horloges est obtenu à partir de la lecture du graphe généré grâce à l'étape de coloration de graphe. Il peut aussi être déduit en combinant la répartition des multiplexeurs de test par signaux *scan_mode* et le rapport *Bypass_KU* créé par le programme en ANNEXE VII. L'attribution des signaux *scan_mode* dans le mini-MIPS sera analysée plus en détail dans le prochain chapitre sur les résultats. On se préoccupe ici uniquement du classement de ces horloges.

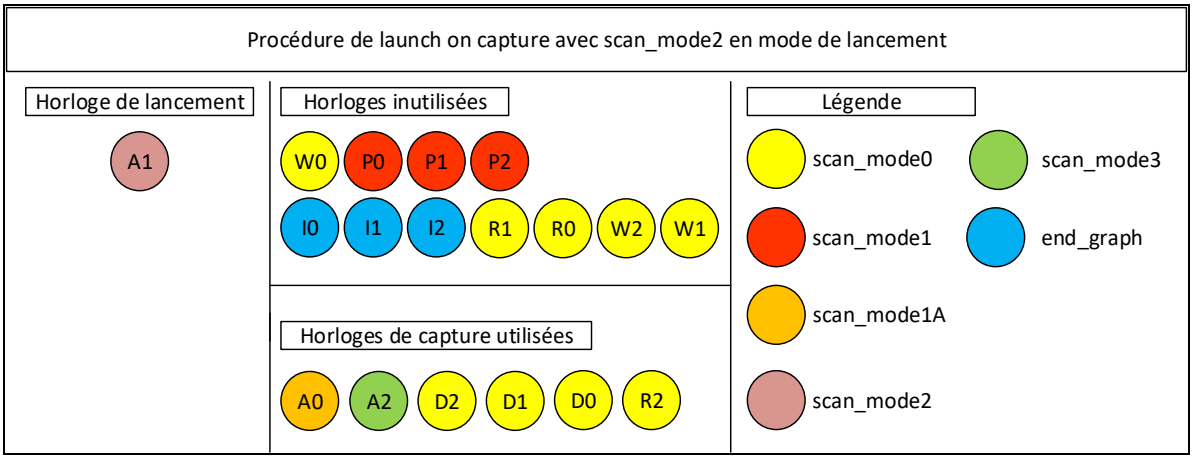

Figure 5.2 Classement des 18 horloges du mini-MIPS par utilité pendant un *launch-on-capture* avec scan_mode_2 en mode de lancement

Dans l'exemple présenté, on observe que certaines horloges appartenant à *scan_mode0* sont utilisées tandis que d'autres sont inutilisées. La distinction entre les deux groupes est très importante pour éviter les erreurs entre la simulation interne de l'ATPG et la simulation externe avec Modelsim. Ces erreurs auraient lieu si l'on avait choisi que toutes les horloges

appartenant à *scan_mode0* soient déclarées dans un même groupe d'horloges et utilisées pour toutes les procédures. Dans le cas de la simulation interne de l'ATPG, le logiciel génère les impulsions de lancement et de capture directement sur les horloges déclarées sans passer par la logique du chemin d'horloge. L'ATPG n'aura donc aucun mal à générer une impulsion de capture à partir d'une impulsion de lancement alors que celles-ci ne sont même pas électriquement reliées. À l'inverse, la simulation de Modelsim simule l'ensemble des chemins d'horloges et peut donc générer les impulsions de capture quand ils sont effectivement reliés électriquement à l'impulsion de lancement. Pour qu'aucune horloge de capture inutilisée ne soit pulsée alors qu'elles ne sont pas reliées à l'horloge de lancement, il faut donc que chaque procédure dispose de ses propres groupes d'horloges. En effet, si dans l'exemple proposé l'horloge R2 est utilisée, ce n'est pas le cas dans la procédure où *scan_mode1* est en mode de lancement où cette fois-ci l'horloge ne doit pas être déclarée.

```
//Horloges pour launch_on_capture avec scan_mode_0 en mode lancement
alias alias_clock0 = "\gen_xu[0].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1 = "\gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3 ="\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
//Horloges pour launch_on_capture avec scan_mode_1 en mode lancement
alias alias_clock0_1 = "\gen_xu[1].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_1 = "\gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
 "\gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
 "\gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
 "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2_1 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_1 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
//Horloges pour launch_on_capture avec scan_mode_2 en mode lancement
alias alias_clock0_2 = "\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y" ,
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_2 = "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2_2 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_2 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
//Horloges launch_on_capture avec scan_mode_3 en mode lancement
alias alias_clock0_3 = "\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_3 = "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2_3 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_3 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
```

Figure 5.3 Définition des horloges extraites du fichier de procédure du mini-MIPS

5.4.2 Définition de la procédure de chargement/déchargement

L'objectif de la définition de la procédure *load_unload* (Figure 5.4) est d'indiquer à l'ATPG comment il doit régler les signaux d'entrée du circuit lors étapes de chargement et de déchargement des vecteurs de test. Dans cette procédure, tous les signaux de *reset* sont désactivés pour ne pas déranger le déroulement du chargement. Le signal *scan_en* est mis à 1, cela permet de déclencher le mode de chargement sur tous les registres à balayage du circuit. Tous les signaux de types *scan_mode* sont réglés à 1, ainsi tous les multiplexeurs de test sélectionnent l'horloge globale de test *scan_clk*. C'est cette horloge qui va permettre le contrôle et le décalage du vecteur de test dans les registres. On observe que ce décalage est appliqué 978 fois, soit le nombre de registres qui constitue la chaine de balayage dans le mini-MIPS.

```
procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
    force scan_reset 0;
    force scan_set 0;
    force alias_allclock 0;
    force scan_clk 0;
    force scan_en 1;
    force scan_mode_0 1;
    force scan_mode_1 1;
    force scan_mode_2 1;
    force scan_mode_3 1;
    force scan_mode_1A 1;
    force end_graph 1;
    force scan_mode_or 1;
    force fsm_mode 0;
    force rst_n 1;
    force rstn_fsm_test 1;
    force test_mode 1;
    force init_TestOrModule_input 0;
   end ;
   apply shift 978;
end;
```

Figure 5.4 Définition de la procédure de chargement/déchargement des vecteurs de test extraite du fichier de procédure du mini-MIPS

5.4.3 Définition des procédures de *launch-on-capture* pour chemins de données convergents

On présente dans cette section une des procédures de *launch-on-capture* utilisées pour le test de chemins de données convergents. Les Figure 5.5 et Figure 5.6 correspondent à la procédure où le signal de contrôle *scan_mode_0* est utilisé en mode de lancement. On observe que la procédure est divisée en deux catégories, le mode interne (mode internal) et le mode externe (mode external).

Le mode interne sera utilisé par l'ATPG pour générer les vecteurs de test tandis que le mode externe est nécessaire pour créer automatiquement le banc de test pour la simulation des vecteurs de test sur Modelsim. L'unique différence entre les deux modes se situe dans cycle de capture. Dans le cas du mode interne (Figure 5.5), on précise quelles horloges l'ATPG doit pulser en tant qu'impulsion de capture avec les instructions (*pulse alias_clock1, pulse alias_clock2*…). Ces instructions sont nécessaires, car comme expliquées dans la section sur la définition des horloges, la génération des vecteurs de l'ATPG ne simule pas le chemin d'horloge du circuit. À l'inverse, dans le mode externe (Figure 5.6), les instructions pulse sont absentes, car, les impulsions de capture seront automatiquement générées par le circuit dès le déclenchement de l'impulsion de lancement.

Le premier cycle de chaque mode correspond à l'étape de lancement. C'est dans cette étape qu'on règle la source d'horloges des multiplexeurs de test avec la configuration des signaux *scan_modes*. On observe que dans l'exemple présenté, seul *scan_mode_0* est réglé en mode de lancement tandis que les autres *scan_modes* sont en mode de capture. Le signal *test_mode* active les modules de contournement des *key_units Bypass_KU* que nous avons vus dans le CHAPITRE 3. *fsm_mode* est mis à 0 pour régler le module de test de machine à états finis en mode de test standard tel qu'expliqué dans le CHAPITRE 3.

```
procedure capture procedure_at_speed_1 =
// Internal mode
  //----------------------------------------
  mode internal =
      // Cycle 1 : launch pulses
    cycle =
     timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0             1;
      force scan_mode_1             0;
      force scan_mode_1A            0;
      force scan_mode_2             0;
      force scan_mode_3             0;
      force end_graph               0;
      force scan_mode_or            0;
      force test_mode               1;
      force fsm_mode                0;
      force rstn_fsm_test           1;
      force init_TestOrModule_input 0;
      force scan_reset              0;
      force scan_set                0;
      force scan_en                 0;
      force rst_n                   1;
      force scan_clk                0 ;
      pulse scan_clk ; // pulse launch clock
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_123 ;
      measure_po;
      pulse alias_clock1 ;
      pulse alias_clock2;
      pulse alias_clock3;
    end;
  end;
```

Figure 5.5 Définition du mode interne de la procédure de *launch-on-capture* avec scan_mode_0 en mode de lancement extraite du fichier de procédure du mini-MIPS

```
  // External mode
  // ( replicates the internal mode without internal clocks )
  //-----------------------------------------
  mode external =
    // Cycle 1 : launch pulses
    cycle =
     timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0              1;
      force scan_mode_1              0;
      force scan_mode_2              0;
      force scan_mode_3              0;
      force scan_mode_1A             0;
      force end_graph                0;
      force scan_mode_or             0;
      force scan_reset               0;
      force scan_set                 0;
      force scan_en                  0;
      force rst_n                    1;
      force scan_clk                 0;
      pulse scan_clk;// pulse launch clock
      force test_mode                1;
      force fsm_mode                 0;
      force rstn_fsm_test            1;
      force init_TestOrModule_input  0;
    end ;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_123;
      measure_po;
    end;
  end ;
end;
```

Figure 5.6 Définition du mode externe de la procédure de *launch-on-capture* avec scan_mode_0 en mode de lancement extraite du fichier de procédure du mini-MIPS

5.4.4 Définition des procédures de *launch-on-capture* pour machine à états finis

Cette section présente un exemple de procédure utilisée pour le test en *launch-on-capture* de machine à états finis (Figure 5.7). Seul le mode interne de la procédure sera décrit ici, on pourra retrouver le mode externe en ANNEXE VI .

Contrairement au *launch-on-capture* sur les chemins de données convergents, cette procédure a besoin d'un cycle de préparation avant l'étape de lancement. C'est dans ce cycle qu'on applique un *reset* sur le module de test de machine à états finis avec la mise à 0 du signal rstn_*FSM_test*. Cette remise à zéro est nécessaire pour que la source d'horloge d'entrée

sélectionne le signal d'horloge *scan_clk* de test global tel qu'expliqué dans le CHAPITRE 3. On remarque que pour éviter de propager l'horloge de lancement aux autres horloges dépendante de celle-ci, tous les signaux *scan_mode* sont réglés en mode de lancement. De cette manière, l'impulsion de lancement reste confinée au sein du domaine d'horloge où la machine à états finis est présente.

Dans le cycle de lancement, le module *FSM_test* est réglé en mode de test de machine à états finis avec les signaux *scan_mode_3* à 0 et *fsm_mode* à 1.

```
procedure capture procedure_at_speed_FSM_A2 =
  mode internal =
    //Cycle 0: Reset FSM_test
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0             1;
      force scan_mode_1             1;
      force scan_mode_2             1;
      force scan_mode_3             1;
      force scan_mode_1A            1;
      force end_graph               1;
      force test_mode               1;
      force scan_mode_or            0;
      force fsm_mode                0;
      force rstn_fsm_test           0;
      force init_TestOrModule_input 0;
      force scan_reset              0;
      force scan_set                0;
      force scan_en                 0;
      force rst_n                   1;
      force scan_clk                0;
    end;
    // Cycle 1 : launch pulses
    cycle =
     timeplate tp_launch_ext ;
     force_pi ;
     force scan_mode_0             1;
     force scan_mode_1             1;
     force scan_mode_2             1;
     force scan_mode_3             0;
     force scan_mode_1A            1;
     force scan_mode_or            0;
     force end_graph               1;
     force test_mode               1;
     force fsm_mode                1;
     force rstn_fsm_test           1;
     force init_TestOrModule_input 0;
     force scan_reset              0;
     force scan_set                0;
     force scan_en                 0;
     force rst_n                   1;
     force scan_clk                0;
     pulse scan_clk;// pulse launch clock
    end ;
   // Cycle 2 : capture pulses
   cycle  =
    timeplate tp_capture_A2;
    measure_po;
    pulse "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
   end;
  end;
```

Figure 5.7 Définition de la procédure de *launch-on-capture* de machine à états finis sur le domaine d'horloge lié au scan_mode_3 extraite du fichier de procédure du mini-MIPS

5.4.5 Définition des procédures de *launch-on-capture* synchrone

La procédure *procedure_sync* présentée en Figure 5.8 permet d'exécuter un *launch-on-capture* de la même manière que dans les circuits synchrones. Pour ce faire, tous les signaux de contrôle *scan_mode* sont réglés à 1 pour que l'horloge globale de test *scan_clk* puisse se propager dans tous les registres. Les deux impulsions de lancement et de capture sont appliquées sur le signal *scan_clk* qui fait office d'horloge globale comme dans les circuits synchrones. Dans cette procédure, tous les registres agissent en tant qu'émetteur et récepteur des impulsions de lancement et capture. Elle couvre donc d'autres chemins de données qui étaient auparavant non couverts, notamment les chemins de catégorie 2 (ICC2D) que nous avons vus dans le CHAPITRE 3. Bien sûr, cette méthode n'est pas aussi valable que les 2 procédures de *launch-on-capture* précédentes, car elle ne teste pas le circuit à sa vitesse nominale (at-speed). En effet, le délai entre l'impulsion de lancement et de capture doit être configuré manuellement dans l'ATPG (ici dans les *timplate tp_launch_ext* et *timeplate tp_capture_all*). Néanmoins, cette procédure nous permettra de savoir quelle est la proportion des chemins de données non couverts par notre configuration de signaux *scan_modes* déterminés dans le programme présenté au CHAPITRE 4.

```
procedure capture procedure_sync =
  mode internal =
    cycle=
     timeplate tp_launch_ext ;
     force_pi ;
     force scan_mode_0             1;
     force scan_mode_1             1;
     force scan_mode_2             1;
     force scan_mode_3             1;
     force scan_mode_1A            1;
     force scan_mode_or            1;
     force end_graph               1;
     force test_mode               1;
     force fsm_mode                0;
     force rstn_fsm_test           1;
     force init_TestOrModule_input 0;
     force scan_reset              0;
     force scan_set                0;
     force scan_en                 0;
     force rst_n                   1;
     force scan_clk                0;
     pulse scan_clk;
    end;
    cycle=
     timeplate tp_capture_all;
     pulse scan_clk;
     measure_po;
    end;
 end;
```

Figure 5.8 Définition de la procédure synchrone de *launch-on-capture* extraite du fichier de procédure du mini-MIPS

5.5 Conclusion

Ce chapitre a été l'occasion de discuter de la phase 4 : génération automatique des vecteurs de test, du flot de testabilité présenté dans le CHAPITRE 4. La première partie a présenté un rappel sur le déroulement des tests. La deuxième partie a présenté l'ensemble des procédures de génération de vecteurs menés sur le panel de circuit du banc d'essai. Elle a aussi montré l'état des signaux *scan_modes* qui contrôlent la source d'horloges des multiplexeurs de test. On a aussi pu constater que les circuits issus de ITC99 et le pipeline typique d'Octasic présentent individuellement un type de test que l'on retrouve dans le mini-MIPS. En effet, le pipeline typique d'Ocastic nous a permis d'éprouver la technique de test des multiplexeurs de test montrée en revue de littérature sans modifications supplémentaire qu'on peut trouver dans le mini-MIPS. Puis les circuits issus du benchmark ITC99 nous ont aidés à développer le test de machine à états finis. Enfin, le mini-MIPS regroupe tous les cas de test des deux

autres circuits tout en apportant de nouvelles difficultés. Pour finir, la troisième partie présente et décrit les étapes les plus importantes de la génération des vecteurs : la définition des horloges, l'étape de chargement et déchargement des vecteurs de test, les *launch-on-capture* sur les chemins de données convergents, les *launch-on-capture* synchrones et les *launch-on-capture* sur les machines à états finis. On a vu à travers un exemple qu'on ne pouvait pas regrouper les horloges par signaux *scan_mode* mais qu'il fallait les décrire pour chaque nouveau test de *launch-on-capture* pour éviter les erreurs entre la simulation de l'ATPG et la simulation externe des vecteurs de test. Tous les résultats générés par ces générations de vecteur seront analysés et discutés dans le prochain chapitre.

CHAPITRE 6

RÉSULTATS ET ANALYSES

6.1 Introduction

Dans ce chapitre, nous présenterons et discuterons des résultats produits par le programme de connexion des signaux de contrôle de test, l'insertion des structures de test et la génération automatisée des vecteurs de test. La première partie sur le programme de connexion des signaux de contrôle sera décomposée en trois sections. La première section présentera les dépendances d'horloge des circuits du banc d'essai détecté par l'algorithme. La deuxième section fera le bilan de l'algorithme de recherche de machine à états finis. Enfin, la troisième section présentera le choix des signaux *scan_mode* déterminé par le programme. Une deuxième partie décrira l'impact de l'insertion des structures de test sur la surface des circuits. Pour finir, on observera et analysera les taux d'efficacité de la détection de panne et les taux de couverture de panne obtenue par la génération automatisée des vecteurs de test à partir du plan d'expérimentation décrit au CHAPITRE 5. Pour conclure, on fera le bilan de ce qu'a apporté ce mémoire en termes de méthode de test et d'automatisation par rapport aux travaux de recherche précédents.

6.2 Programme de connexion des signaux de contrôle de test

Dans cette section, on présente les résultats intermédiaires du programme de connexion des signaux de contrôle présenté dans le CHAPITRE 4. On s'intéresse en particulier aux résultats de 3 algorithmes : l'analyse de la dépendance des horloges, la détection des machines à états finis et la détermination du nombre et de la connexion des signaux *scan_mode*. L'observation et l'analyse de ces résultats pourront nous aider à comprendre les futures modifications qui pourraient être introduites dans le flot de testabilité automatisé pour améliorer la prise en charge des circuits étudiés. On étudiera ici seulement le pipeline typique de la topologie d'Octasic et le mini-MIPS asynchrone puisque les circuits issus du benchmark ITC99

n'emploient pas le programme de connexion des signaux de contrôle dans leur flot de testabilité.

6.2.1 Analyse de la dépendance des horloges

On a vu dans le CHAPITRE 4 que l'algorithme d'analyse de la dépendance des horloges était chargé d'établir la dépendance entre toutes les horloges du circuit. On présente dans cette partie les résultats de cet algorithme sous la forme de matrice de dépendance. Ainsi, dans les tableaux 6.1 et 6.2, les horloges émettrices sont représentées en colonne tandis que les horloges réceptrices qui sont dépendantes des horloges émettrices pour générer leur impulsion sont représentées en ligne. La valeur 1 identifie une connexion entre l'horloge émettrice et réceptrice tandis que le 0 représente l'absence de connexion. Les cases sont colorées en vert lors d'une bonne détection de la part de l'algorithme par rapport à la situation réelle. Les cases colorées en rouge identifient de fausses détections trouvées par l'algorithme.

6.2.1.1 Pipeline asynchrone typique d'Octasic

On observe sur le Tableau 6.1 les résultats de la dépendance des horloges pour le pipeline asynchrone typique d'Octasic. Toutes les connexions trouvées par l'algorithme correspondent à la situation réelle. Aucune fausse détection n'est à déplorer.

Tableau 6.1 Résultat de l'algorithme de l'analyse de la dépendance des horloges pour le pipeline asynchrone typique d'Octasic

		Horloges réceptrices			
		M0	M1	M2	M3
Horloges émettrices	M0	0	1	0	0
	M1	0	0	1	0
	M2	0	0	0	1
	M3	0	0	0	0

L'algorithme développé semble efficace pour les circuits comme le pipeline typique d'Octasic puisqu'on a constaté qu'aucune erreur n'est à déplorer. La vérification de ce résultat peut se faire en observant la dépendance des horloges sur le schéma du circuit qui est visualisable à la Figure 2.3 du CHAPITRE 2.

6.2.1.2 Mini-MIPS

Le Tableau 6.2 représente les résultats de la dépendance des horloges pour le mini-MIPS asynchrone. On observe que 99 connexions détectées n'existent pas en réalité tandis que 36 connexions sont correctes.

Tableau 6.2 Résultat de l'algorithme de l'analyse de la dépendance des horloges pour le mini-MIPS

		Horloges réceptrices																	
Horloges émettrices		R_0	A_0	D_0	W_0	P_0	I_0	R_1	A_1	D_1	W_1	P_1	I_1	R_2	A_2	D_2	W_2	P_2	I_2
	R_0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
	A_0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0
	D_0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
	W_0	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0
	P_0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1
	I_0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0
	R_1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
	A_1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0
	D_1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
	W_1	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0
	P_1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1
	I_1	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0
	R_2	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
	A_2	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0
	D_2	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
	W_2	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0
	P_2	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1
	I_2	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	0	1	0

L'analyse de la dépendance des horloges s'avère beaucoup plus compliquée dans le cas d'un circuit complexe comme le mini-MIPS. On a constaté que la majorité des dépendances

détectée sont erronées. Ces erreurs s'expliquent par la présence des *key units* dans le réseau d'horloges du circuit. En effet, les mauvaises connexions se construisent lors de l'étape de la contraction du graphe qui supprime tous les éléments de la netlist excepté les multiplexeurs de test. Par le fait que tous les *key units* sont reliés entre eux par un signal *key* de type jeton, des domaines d'horloges qui ne sont pas directement dépendants se retrouvent connectés ensemble. Pour illustrer nos propos, on prend comme exemple les *key units* D2, D1 et D0 (Figure 6.1). Si l'on se concentre sur le multiplexeur D0, celui-ci est théoriquement uniquement dépendant de A0 et W2, car ce sont ces horloges qui contrôlent le *key units* associé à D0. Mais lorsque le programme exécute l'algorithme de contraction de graphe, A2 et W1 se retrouvent connectés à D0 par l'intermédiaire du *key unit* D2 qui est relié au *key unit* D0. Le même phénomène se produit avec A1 et W0 par l'intermédiaire des *key units* D1 et D2. En résumé, le programme estime que les horloges réceptrices sont dépendantes de toutes les horloges reliées au *key units* qui contrôle la même ressource. C'est la cause principale de la confusion des chemins de données convergents et indirectement convergents de catégorie 1. On discutera de l'impact cette confusion dans la section 6.2.3.2 et on proposera des pistes de résolutions dans les recommandations.

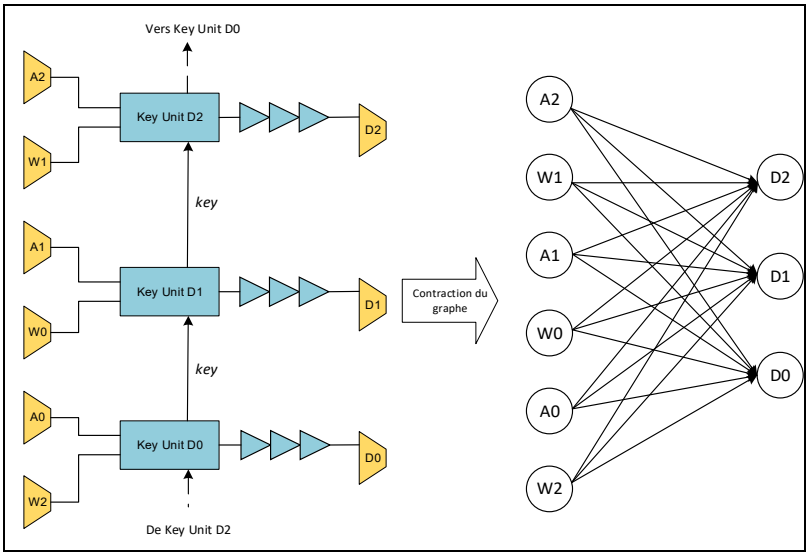

Figure 6.1 Illustration des résultats de l'algorithme de contraction de graphe sur un groupe de key unit qui contrôle la même ressource

6.2.2 Recherche de machine à états finis

La détection des machines à états finis est basée sur le graphe contracté généré par l'analyse de la dépendance des horloges. Si un sommet qui représente un domaine d'horloge boucle sur lui-même, cela signifie qu'il contrôle des machines à états finis. Le Tableau 6.3 présente le résultat de cette détection.

Tableau 6.3 Résultat de l'algorithme de détection des machines à états finis

Circuits	Domaine d'horloge contrôlant des machines à états finis détectés
Pipeline typique d'Octasic	Aucun
Mini-MIPS	A0, A1, A2

Aucune machine à états finis n'est détectée sur le pipeline typique d'Octasic et c'est tout à fait normal puisque le circuit n'en possède pas

L'algorithme détecte 3 domaines d'horloge A0, A1, A2 qui contrôlent des machines à états finis dans le mini-MIPS. Or, on sait que ce ne sont pas les seuls domaines d'horloge qui en contrôle. En effet, l'inspection du schéma nous a permis de constater que d'autres machines à états finis sont présentes, notamment dans le compteur de programme. L'algorithme aurait donc dû aussi détecter P0, P1 et P2.

L'analyse de ces résultats nous a permis de nous rendre compte que l'hypothèse de départ (p. 91) à la base de cet algorithme est fausse dans le mini-MIPS. Elle est uniquement vraie dans les circuits issus du benchmark ITC99. En effet, comme ces circuits ne possèdent qu'un domaine d'horloge, il boucle nécessairement sur eux-mêmes pour s'autoalimenter, mais ce n'est pas le cas des horloges dans le mini-MIPS qui sont au nombre de 18. Dans une certaine mesure, les horloges du mini-MIPS bouclent toutes sur elles-mêmes puisqu'elles emploient

une structure d'horloge en anneau tel que constaté dans le CHAPITRE 2. Mais, en aucun cas, un domaine d'horloge ne boucle directement sur lui-même.

Les bonnes détections de l'algorithme sont en fait des « faux positifs ». En effet, les *key units* qui contrôle l'ALU (A0, A1, A2) sont équipés d'un mécanisme de *stall* qui permet de mettre en pause ces unités. On peut d'ailleurs l'observer dans le schéma Figure 3.12 du CHAPITRE 3. Ces registres chargés du *stall* sont en partie contrôlés par les domaines d'horloges du même groupe. Et par le même phénomène que celui présenté dans l'analyse de la dépendance des horloges, l'horloge se retrouve connectée à elle-même avec l'algorithme de contraction de graphe. Si l'on prend comme exemple A2, il se retrouve connecté à lui-même par l'intermédiaire des *key units* A0 et A1 comme le montre la Figure 6.2. Dans notre cas ces faux positifs correspondent à de vraies machines à états finis et cette erreur n'a donc pas d'impact négatif sur la génération automatique des vecteurs de tests du mini-MIPS. On proposera une piste de résolution de cet algorithme dans les recommandations.

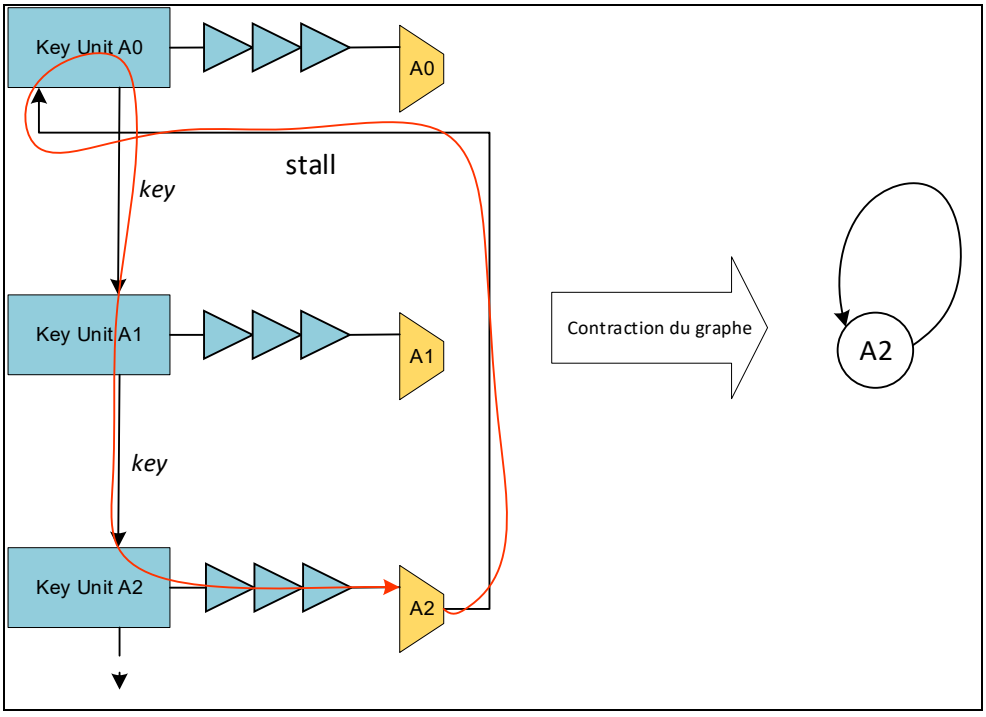

Figure 6.2 Illustration de l'effet du mécanisme de stall des key units sur l'algorithme de contraction de graphe

6.2.3 Détermination du nombre et de la connexion des signaux *scan_mode*

Les résultats présentés dans cette section sont obtenus à partir de la combinaison de trois algorithmes : l'analyse de la dépendance des horloges, la recherche de chemins de données convergents et la coloration de graphe pour déterminer les signaux *scan_mode*. Ces résultats sont visualisables de sous la forme de graphe généré à l'aide de Graphviz. Chaque sommet du graphe est l'équivalent d'un multiplexeur de test qui représente un domaine d'horloge du circuit. Lorsqu'un chemin de donnée est présent entre deux domaines d'horloge, on représente ce chemin par une arrête dont la direction est la même que celle du chemin. Les domaines d'horloge sont colorés selon leurs signaux de contrôle *scan_mode* associé.

6.2.3.1 Pipeline asynchrone typique d'Octasic

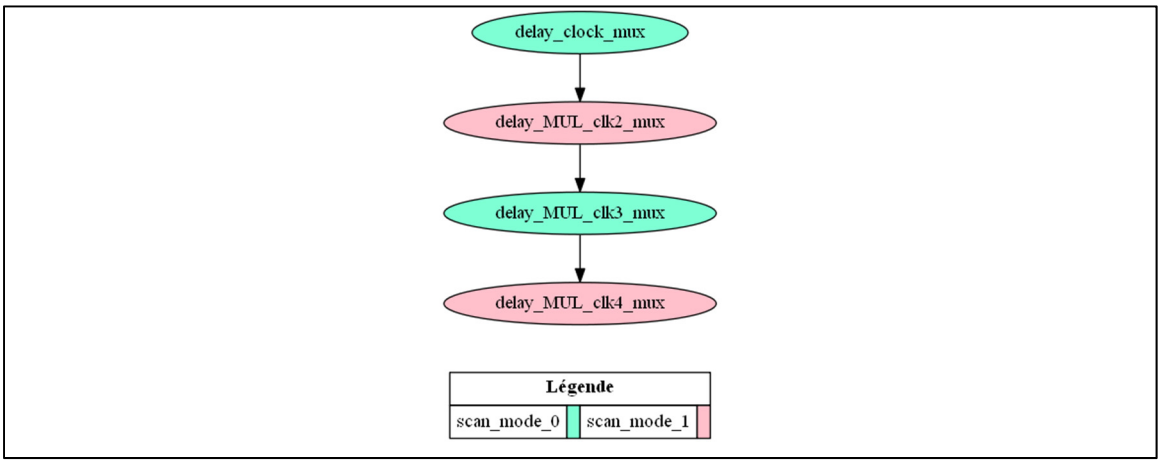

Figure 6.3 Graphe représentant les chemins de données convergents existant entre les domaines d'horloges du pipeline asynchrone typique d'Octasic

Les résultats du pipeline asynchrone typique d'Octasic (Figure 6.3) sont bien ceux attendus. Tous les chemins de donnée ont été détectés et le programme a trouvé qu'il fallait 2 signaux scan_mode pour couvrir toutes les pannes. Le résultat correspond parfaitement au cas présenté en revue de littérature. Le circuit ne présente aucune difficulté pour le programme,

chaque domaine d’horloge est uniquement dépendant d’un seul autre domaine d’horloge et tous les chemins de donnée du circuit sont convergents.

6.2.3.2 Mini-MIPS

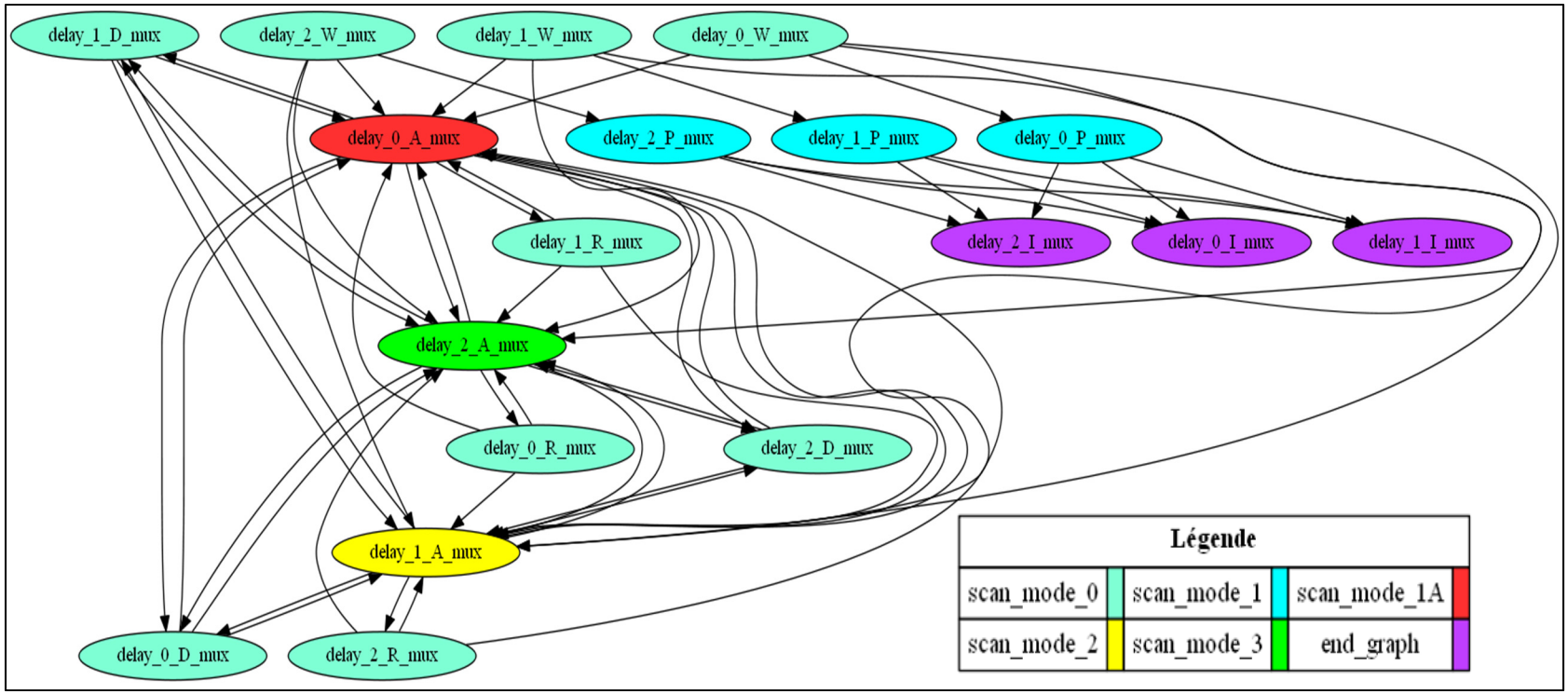

Figure 6.4 Graphe représentant les chemins de données convergents existant entre les domaines d'horloge du mini-MIPS

Les résultats du mini-MIPS sont eux plus contrastés (Figure 6.4). On constate que le programme a trouvé que 6 signaux *scan_mode* sont nécessaires pour couvrir tous les chemins de données convergents. En réalité, le nombre de signaux de contrôle pourrait être abaissé à 4.

On a vu dans l'analyse de la dépendance des horloges qu'une bonne partie des dépendances d'horloge détectée sont fausses. L'algorithme de recherche des chemins de donnée convergents débute donc avec de mauvaises informations, puisqu'il considère qu'il existe beaucoup plus de dépendance qu'en réalité. Cela a pour conséquence l'inclusion des chemins de données non convergents de catégorie 1 dans le graphe final. En effet, ces chemins sont considérés comme convergents par le programme à cause des erreurs produites par la contraction du graphe. Cette confusion est embêtante puisque l'objectif de départ était de permettre le test des chemins convergents tel que présenté dans la revue de littérature.

De plus, les registres du mécanisme de *stall* des *key units* associé à l'ALU sont aussi confondus en chemin de donnée convergent par l'algorithme. C'est pour cette raison que les sommets représentant les domaines d'horloges des ALU ont 11 horloges en entrée (2 chemins convergents, 4 chemins de type ICC1D, 6 chemins du au mécanisme de *stall*). Le nombre de signaux de type *scan_mode* pourrait donc encore être diminué si toutes ces erreurs n'étaient pas présentes. Par exemple, tous les domaines d'horloges correspondant aux ALU (delay_0_A_mux, delay_1_A_mux, delay_2_A_mux) pourraient avoir le même type de signal *scan_mode*, on tomberait donc à 4 signaux de contrôle au lieu de 6.

L'avantage de cette inclusion permet cependant à la procédure de test de couvrir plus de pannes puisqu'on ajoute les chemins de type ICC1D en plus des chemins de donnée convergents. Le taux de la couverture de panne va donc inévitablement augmenter. Mais d'un autre côté, on a vu dans le CHAPITRE 3 que le test des chemins ICC1D n'est pas valable parce qu'ils sont testés plus rapidement que la vitesse nominale. On se trouve donc dans une situation d'*over-test*, c'est-à-dire que l'on teste des chemins de donnée à une fréquence plus haute que la vitesse nominale. L'*over-test peut* avoir pour conséquence de

considérer un circuit comme défectueux, car il n'est pas conçu pour fonctionner plus rapidement que la vitesse nominale. En revanche si aucune panne n'est détectée, on s'assure que le chemin ICC1D fonctionne bien.

6.3 Surface des structures de test

Cette section a pour but de faire le bilan de l'impact de l'insertion des structures de test sur la surface du circuit. Le Tableau 6.4 résume l'impact de l'insertion des structures de test dans les circuits du banc d'essai.

Avec 50.4%, 53.02% et 50.9%, les circuits issus du benchmark ITC99 sont ceux qui ont été les plus impactés sur leur surface à cause de l'insertion des structures de test.
La surface du pipeline typique d'Octasic est particulièrement très peu impactée par l'insertion des structures de test. En effet, on constate une augmentation de la surface de circuit de seulement 2.97%.

Enfin le mini-MIPS voit sa surface augmenter de 16.3%. On peut noter que les diverses erreurs des algorithmes que nous avons précédemment discutées ont un impact négligeable sur la surface des structures de test. En effet, ces erreurs se traduisent uniquement par une augmentation minime du nombre de fils dans le circuit, mais en aucun cas par un ajout de structures de test.

Tableau 6.4 Impact de l'insertion des structures de test sur la surface des circuits

Circuits	Avant l'insertion des structures de test			Après l'insertion des structures de test			Taux de la surface nécessaire supplémentaire pour implémenter le test
	Porte	Registre	Surface (μm^2)	Porte	Registre	Surface (μm^2)	
ITC99-b01	73	6	614	103	7	924	50,40%
ITC99-b02	64	5	579	94	6	886	53,02%
ITC99-b06	67	9	624	97	10	942	50,90%
Pipeline typique d'Octasic	102	17	1007	132	17	1037	2.97%
Mini-MIPS asynchrone	3573	997	26406	3896	1000	30711	16,3%

L'augmentation importante de surface sur les circuits issus du benchmark ITC99 s'explique par le fait qu'ils se sont fait insérer le module de machine de test de machine à états finis *FSM_test.* Or on sait que ce module représente une surface de 264 μm^2, ce qui représente en moyenne environ 42% de la taille du circuit initial et 84.7% de la surface des structures de test insérées dans les circuits. Le reste de la surface nécessaire est à attribuer au remplacement des registres standard par les registres à balayage. Cette augmentation de surface importante devrait donc être moindre lors de l'insertion du module *FSM_test* sur des machines à états finis beaucoup plus complexe puisque le module de test représente un coût constant en surface.

On a observé que la surface du pipeline typique d'Octasic est peu impactée par l'insertion des structures de test. En effet, hormis l'insertion des registres à balayage, les 4 multiplexeurs à 2 entrées sont les seules structures de test supplémentaire insérées.

Le mini-MIPS asynchrone est le plus représentatif de l'impact de la technique de tests sur la surface du circuit. On rappelle qu'en plus des registres à balayage, 3 modules *FSM_test*, 18 modules *Bypass_KU* et 15 multiplexeurs ont été insérés dans le circuit. Ce qui résulte d'un accroissement de la surface du circuit de 16.3%. À titre de comparaison, la technique de test basé sur les multiplexeurs de Beest (te Beest & Peeters, 2005) arrive à un agrandissement de la surface du circuit de 25% sur un microprocesseur 80C51 asynchrone. On peut donc considérer l'impact de notre technique de test comme assez faible sur la surface du circuit.

6.4 Qualité du test

Pour chaque circuit du banc d'essai, cette section présente les résultats des indicateurs de mesure de la qualité du plan d'expérimentation détaillé dans le CHAPITRE 5. La première partie discute du taux d'efficacité de la détection de panne (équation 1.2) tandis que la seconde partie analyse le taux de couverture de panne (équation 1.1) de chacun des circuits. Le Tableau 6.5 présente les résultats du plan d'expérimentation détaillé au CHAPITRE 5.

Tableau 6.5 Résultat des taux de couverture de pannes et taux d'efficacité de la détection de pannes de la génération automatisée à partir du plan d'expérimentation présenté dans le chapitre précédent

Circuits		**Test**	**Taux d'efficacité de la détection de pannes**	**Taux de couverture de panne**
ITC99	b01	*Launch-on-capture* FSM	100%	66,42%
	b02	*Launch-on-capture* FSM	100%	75%
	b06	*Launch-on-capture* FSM	100%	68,61%
Pipeline typique d'Octasic		Collé-à	100%	95,64%
		Launch-on-capture à vitesse nominale	100%	84,36%
Mini-MIPS		Collé-à	100%	89,43%
		Launch-on-capture à vitesse nominale	98,87%	76,08%
		Launch-on-capture partiellement à vitesse nominale	98,81%	81,50%

6.4.1 Taux d'efficacité de la détection de panne

Dans une première lecture du Tableau 6.5, on peut remarquer que tous les taux d'efficacité de la détection de pannes excepté les *launch-on-capture* du mini-MIPS sont à 100%. On rappelle que le taux d'efficacité de la détection de panne retranche les pannes indétectables relatives au modèle de test choisi. Il nous indique donc l'efficacité des procédures de test exécutées par rapport à l'emplacement des structures de test. On peut donc conclure de ce résultat que la majorité de nos procédures de test sont complètes. Les taux d'efficacité des *launch-on-capture* du mini-MIPS qui sont de 98.87% et 98.81% nous indiquent qu'on

pourrait encore ajouter une ou plusieurs procédures pour augmenter notre couverture de panne.

6.4.2 Taux de couverture de pannes

6.4.2.1 ITC99

Les circuits formés du générateur d'horloge asynchrone et des circuits b01, b02 et b06 obtiennent respectivement des taux de couverture de pannes très proche de ceux constatés dans (Touati, 2016) présenté dans le CHAPITRE 2. On rappelle que ces taux étaient respectivement 69.91%,82% et 69.82% pour les circuits b01, b02 et b06. Nos résultats sont de 66.42%, 75% et 68,61% pour l'ensemble du circuit formé du générateur d'horloge asynchrone et des circuits b01, b02 et b06. Si l'on s'intéresse uniquement aux taux de couverture de panne des circuits b01, b02 et b06 tout en omettant le générateur d'horloge, on obtient 69.35%, 79.89% et 70.83%. Ces résultats sont donc encore plus proches de ceux présentés au CHAPITRE 2 et même encore plus élevés pour b06. Les petites variations entre nos résultats et ceux de (Touati, 2016) peuvent s'expliquer par le fait que nous n'utilisons ni la même technologie, ni le même outil de synthèse, les netlists testés sont donc différentes. On peut donc conclure par ces résultats que notre technique de test de machine à états finis à vitesse nominale est pratiquement aussi efficace qu'un test de transition *launch-on-capture* synchrone.

6.4.2.2 Pipeline typique d'Octasic

On remarque sur le Tableau 6.5 que les tests collés-à et *launch-on-capture* à vitesse nominale obtiennent respectivement des taux de couverture de panne de 95.64% et 84.36%. Durant le test collé-à, les signaux *scan_mode* des 4 multiplexeurs de test sont contraints à 1 pour que l'ATPG les contrôle. On retrouverait sans ces contraintes un taux de couverture de panne beaucoup plus fréquent de 99% auquel le modèle collé-à nous a souvent habitué dans la littérature pour les circuits synchrones. Quant au taux de couverture de panne de 84.36%

obtenue par le test de *launch-on-capture*, il est tout à fait cohérent. En effet, certaines pannes sont indétectables par le modèle de test *launch-on-capture* par rapport au modèle collé-à.

6.4.2.3 Mini-MIPS

On remarque que la procédure de test employant le modèle collé-à nous permet d'atteindre un taux de couverture de panne 89.43%. C'est le taux maximum réalisable par la disposition de structure de test présenté dans ce mémoire. On en déduit qu'il reste 10.57% de panne indétectable dans le circuit. L'analyse du rapport des pannes restantes nous permet de constater qu'elles proviennent majoritairement des structures de gestion de l'horloge, notamment les *key units* qui sont tous contournés durant les tests. Une partie de ces pannes vient aussi des structures de test insérées. Enfin, comme le pipeline typique d'Octasic, certaine entrée sont contrainte durant les tests et ne peuvent donc pas être contrôlé par l'APTG, cela contribue a augmenté les pannes indétectables.

La procédure de test de *launch-on-capture* partiellement à vitesse nominale est un mélange de *launch-on-capture* qui tire parti des lignes à délai et de *launch-on-capture* synchrones. Le taux de couverture de panne obtenue est de 81.50%. La différence avec le test de type collé-à est de 7.93%. La comparaison des deux rapports de test nous permet de déterminer la provenance des pannes restantes. On observe qu'environ 27.7% des pannes restantes de cette procédure correspondent aux signaux de réinitialisation de tous les registres du circuit. On peut donc dire qu'environ $(100\% - 81.42\%) * 27.7\% = 5.12\%$ des 7.93% de différence entre le test collé-à et cette procédure sont dus aux signaux de réinitialisation.

Les 2.81% restant sont dû à de la logique combinatoire sans registre de capture (Figure 6.5). Tel que nous l'avons vu dans le CHAPITRE 2, durant la conversion du design du mini-MIPS du FPGA vers la librairie ASIC 45nm de Cadence, nous avons fait le choix d'externaliser tous les modules contenant de la mémoire. Ainsi, la banque de registre, la mémoire d'instruction et la mémoire vive du processeur sont externalisées dans le banc de test du circuit. Ce processus d'extraction de ces modules de mémoire a eu pour effet de supprimer

les registres de capture du *crossbar-switch* du circuit. Les pannes restantes sont donc dues à la logique combinatoire associée à l'accès des modules de mémoire présente dans le *crossbar-switch*.

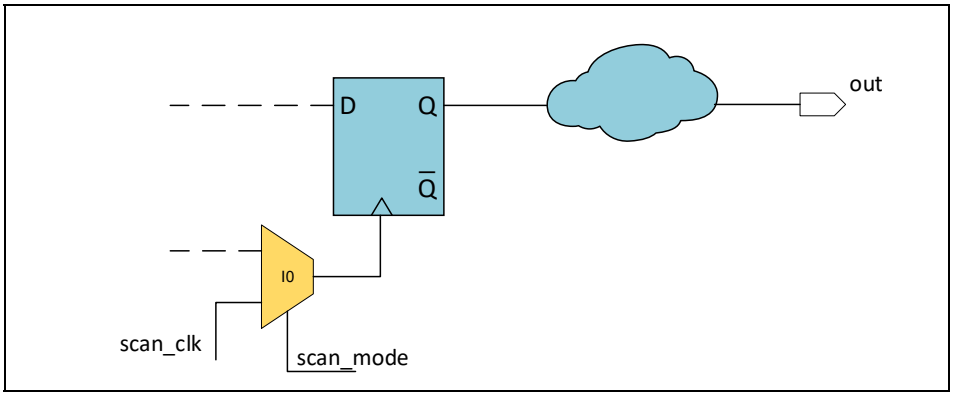

Figure 6.5 La logique combinatoire est directement reliée à la sortie sans registres de capture, ce qui rend ses pannes indétectables en *launch-on-capture*

La procédure de *launch-on-capture* à vitesse nominale tire uniquement parti des lignes à délai pour exécuter les tests. Elle permet d'atteindre un taux de couverture de panne de 76.08%, soit une différence de $81.50\% - 76.08\% = 5.42\ \%$ avec la procédure de *launch-on-capture* partiellement à vitesse nominale. On retrouve 3 types de pannes à l'origine de cette différence.

Le 1er type concerne les chemins de donnée non convergents de catégorie 2. En effet, contrairement aux ICC1D qui sont confondus avec les chemins convergents par le programme ce n'est pas le cas des ICC2D qui ne sont donc pas testés.

Le 2ème type concerne les chemins de donnée adoptant une topologie synchrone. Les pannes de ce type de chemin ne peuvent être logiquement que détectées par une procédure de *launch-on-capture* synchrone qui n'est pas présente dans la procédure de *launch-on-capture* à vitesse nominale.

Le 3ème type concerne les machines à états finis qui n'ont pas été détectées par notre logiciel parce que leurs horloges ne bouclent pas sur elle-même de manière apparente. Comme le deuxième type, ces pannes restantes peuvent être détectées par une procédure de *launch-on-capture* synchrone.

On résume dans le Tableau 6.6, l'ensemble des pannes restantes dans le mini-MIPS après l'exécution de tous les *launchs on capture* à vitesse nominale.

Tableau 6.6 Résumé et provenance des pannes restantes dans le test entièrement à vitesse nominale

Taux de panne	État des pannes	Provenance de la panne
76,08%	détectées	--
10,57%	non détectées	Structure de test, circuit d'horloge, entrées contraintes
5,12%	non détectées	Signaux de reset des registres
2,81%	non détectées	Logique combinatoire dans le crossbar-switch sans registres de capture
5,42%	non détectées	Chemin ICC2D, chemin de donnée synchrone, machines à états finis non détectées

6.5 Évolution des méthodes de tests et de leurs automatisations par rapport aux précédents travaux

Cette section a pour objectif de montrer l'évolution des méthodes de tests et de leurs automatisations. On compare les statuts de ces méthodes avant et après ce mémoire dans le Tableau 6.7 pour chaque circuit du banc d'essai. La grande colonne centrale du tableau fait référence aux travaux de (Hasib et al., 2018) pour sa méthode de test initiale et (TÊTU, 2014)

pour son travail sur la recherche des points de convergence. La dernière grande colonne de gauche fait le constat de l'état de ces méthodes et leur automatisation après ce mémoire.

Tableau 6.7 Comparaison de l'évolution de l'automatisation entre les précédentes méthodes et ce mémoire

	Hasib, Têtu		**Ce mémoire**	
Circuit	Méthode	Automatisation	Méthode	Automatisation
Pipeline typique d'Octasic	✓	limitée	✓	✓
ITC99	✗	✗	✓	✗
Mini-MIPS	✗	✗	✓	limitée

On constate que seul le pipeline typique d'Octasic est supporté avant les travaux réalisés dans le cadre de ce mémoire. La méthode de test de Hasib permet uniquement de tester les circuits dont l'architecture est strictement identique au pipeline d'Octasic. L'automatisation de cette méthode était auparavant considérée comme « limitée » car TÊTU avait uniquement travaillé sur la recherche des points de convergences qui est une étape parmi beaucoup d'autres pour permettre une automatisation totale de la méthode de test. Les travaux de ce mémoire ont donc permis d'apporter cette automatisation pour les circuits pipeline typiques d'Octasic. De plus, aucune méthode de test n'existait pour les machines à états finis issus du jeu de circuit ITC99 et le mini-MIPS asynchrone. On a vu dans ce mémoire qu'une méthode de test a été créée pour les circuits d'ITC99, mais que son automatisation totale n'a pour l'instant pas été investigué. Enfin, contrairement aux précédents travaux, les recherches de ce mémoire supportent les circuits dérivés de l'architecture d'Octasic comme le mini-MIPS. Son automatisation reste toutefois encore « limitée » car certaines étapes font défaut, notamment la détection des machines à états finis.

6.6 Conclusion

Dans ce chapitre, nous avons présenté et discuté des résultats produits par le programme de connexion des signaux de contrôle de test, l'insertion des structures de test et la génération automatisée des vecteurs de test.

Nous avons constaté que malgré les premiers efforts développés dans le programme de connexion des signaux de contrôle, certaines erreurs d'analyse subsistent. Ainsi, on a vu que l'analyse de la dépendance des horloges détecte beaucoup de fausses relations entre les horloges lors de la gestion de circuit complexe comme le mini-MIPS. Ces erreurs se répercutent sur l'ensemble des algorithmes et ont trois conséquences sur les résultats du programme :

1. Le programme confond les chemins de données convergents et non convergents de catégorie 1. Il en résulte donc que notre méthode de test est sujette à l'*over-testing*, c'est-à-dire que l'on peut considérer les circuits comme défectueux, car ils sont testés plus rapidement que la vitesse nominale.
2. L'algorithme de détection de machine à états finis ne fonctionne pas. Les quelques détections qui s'apparentent comme bonnes (A0,A1,A2) sont le fruit d'erreurs de l'algorithme tels qu'expliqué dans la section 6.2.2.
3. Le nombre de signaux de contrôle *scan_mode* est accru. Dans notre cas cela n'a pas d'importance, car on aurait tout de même eu besoin de signaux individuels pour contrôler le test des machines à états finis. Le nombre plus important de signaux *scan_mode* a pour effet de diminuer la rapidité du test, mais n'affecte en aucun cas sa qualité.

De plus, l'analyse de l'impact de l'insertion des structures de test sur la surface des circuits nous a permis de constater que notre technique de test était plutôt économique par rapport à d'autres méthodes de test i.e., (te Beest & Peeters, 2005).

Enfin, les taux de couverture de panne obtenue par la génération automatisée des vecteurs de test sont plutôt satisfaisants, car on a vu que les pannes subsistantes étaient soit voulues ou non-testables avec un modèle de panne de délai.

CONCLUSION

L'objectif principal de ce mémoire était d'améliorer et d'automatiser le processus d'insertion des structures de test pour les circuits asynchrones employant la topologie d'Octasic. La revue de littérature nous a permis de constater qu'il existe une grande variété de technique de conception asynchrone et que chacune de ces techniques possède son propre flot de testabilité. La méthode d'Octasic étant singulière, nous avons nous aussi dû créer notre propre procédé. Nous avons aussi pu voir que même si l'article fondateur de ces travaux (Hasib et al., 2018) définit globalement la technique de test utilisé dans ce mémoire, la gestion de certains cas particuliers comme le test de machines à états finis n'était pas caractérisée. De plus, la technique avait été testée sur des circuits relativement simples par rapport au mini-MIPS asynchrone.

On a aussi vu que des travaux préliminaires (TÊTU, 2014) avaient déjà tenté de créer un outil capable de trouver les points de convergences des circuits typiques d'Octasic. Une étape similaire à notre recherche des chemins de données convergents. Cependant, même si cet outil trouvait tous les points de convergence, on a vu que son temps d'exécution, de l'ordre de 60 jours pour des circuits peu complexes (~40 registres, 3 domaines d'horloges) comparativement à notre mini-MIPS (997 registres, 18 domaines d'horloges) était beaucoup trop important pour être utilisable. En comparaison, le programme développé dans ce mémoire tente de répondre à un problème plus complexe en seulement une dizaine de minutes sur une configuration matérielle similaire aux travaux de TÊTU.

Cependant, on a pu voir dans le CHAPITRE 6 que la solution proposée ici comporte aussi son lot de problèmes. Le manque d'information disponible dans la netlist de départ et le choix des algorithmes ont conduit le programme développé à détecter beaucoup plus de chemin convergent qu'il en existe en réalité. Ces erreurs s'expliquent aussi par l'architecture singulière du mini-MIPS, le circuit a été développé avec des méthodes automatisées visant à faciliter sa conception, mais qui complique sa testabilité. On pense notamment à sa structure d'horloges en anneaux qui implique beaucoup de registre et de logique combinatoire au sein

du réseau d'horloges, mais aussi à la variété des chemins de données que propose le mini-MIPS asynchrone. On a pu voir dans le CHAPITRE 3 que le processeur ne comporte pas moins de 8 chemins de données différents alors que la topologie d'Octasic n'en prévoit normalement qu'un, le chemin de donnée convergent. Le mini-MIPS s'éloigne donc drastiquement de l'architecture d'Octasic et ne peut donc être entièrement compatible avec la méthode de test initialement développé.

Les améliorations qui ont été apportées par ce mémoire par rapport à la technique initiale sont tout de même importantes. On a d'abord constitué un flot de testabilité fixe qui pourrait être utilisé pour n'importe quel circuit qui emploie les mêmes techniques de conception asynchrone. Une partie de ce flot, plus particulièrement l'insertion des structures de test spécifique, est presque totalement automatisée puisqu'il ne reste plus qu'au concepteur à identifier les horloges du circuit. Contrairement à la méthode initiale, notre processus prend en charge le test de machines à états finis bien que leur détection reste un des points à améliorer. Enfin, le procédé a été testé sur un circuit dont la structure du réseau d'horloges est plus complexe que sur le PicoAlu et le multiplieur testé dans l'article initial (Hasib et al., 2018). Pour finir, les taux de couverture des pannes à vitesse nominale avec l'ATPG obtenue grâce à ce flot sont satisfaisants puisqu'ils comprennent l'ensemble des chemins de donnée définis par la topologie de design d'Octasic.

RECOMMANDATIONS

Même si les travaux de ce mémoire ont fait progresser le flot de testabilité des circuits qui emploie la topologie de design d'Octasic, les résultats ne sont pas parfaits et l'on peut donc recommander quelques améliorations.

Le plus gros problème concerne la confusion par le programme des chemins de données convergent et ICC1D. On a vu que ces erreurs mènent l'ingénieur chargé des tests à tester des chemins non convergents plus rapidement que la vitesse nominale. Le CHAPITRE 6 nous a révélé que le problème se situait dans l'algorithme qui définit la dépendance des horloges et en particulier la partie sur la contraction de graphe. Lors de cette contraction, les structures complexes du réseau d'horloges du mini-MIPS qu'on appelle *key unit* sont supprimées et tous leurs voisins d'entrée sont connectés au voisin de sortie. Cette opération serait valable si uniquement des horloges d'entrée et de sortie étaient connectées à ces *key-units*, mais en réalité d'autres signaux de contrôle comme les *stalls* ou le signal de type jeton sont présents. Il faudrait donc pouvoir identifier les *key-units* et les signaux de contrôle pour empêcher les mauvaises connexions de se réaliser dans le graphe lors de l'opération de contraction. Or la netlist actuellement utilisée en entrée du programme ne possède aucune information sur la hiérarchie du circuit puisqu'elle est complètement aplatie. Une des améliorations pourrait donc être de garder la hiérarchie du design lors de la synthèse pour pouvoir labéliser les portes logiques selon leurs appartenances aux différents modules dans le graphe initial. Ainsi, on pourrait créer un graphe de plus haut niveau où nous pourrions identifier les structures complexes et réaliser une analyse plus globale du circuit pour éviter la confusion initiale. Cette modification du programme permettrait d'empêcher la confusion des chemins de données convergents et ICC1D. Elle diminuerait donc le nombre de connexions nécessaires sur les modules de contournement des *key-units Bypass-ku,* ce qui aurait directement un impact sur le nombre de signaux *scan_mode* nécessaires. L'effort nécessaire à l'implémentation de ces changements est élevé, car ils demanderaient une modification importante de la représentation des graphes dans le programme.

Un second problème est la détection des machines à états finis qui ne fonctionne tout simplement pas pour les circuits plus complexes comme le mini-MIPS. On a vu que ces mauvaises détections sont dues à l'hypothèse prise au départ, c'est-à-dire que l'on considérait qu'un domaine d'horloge contrôle une FSM lorsqu'un retour sur lui-même est présent dans le graphe de dépendance des horloges. À la place de cette hypothèse, on pourrait plutôt identifier les registres qui présentent un retour sur eux-mêmes. En effet, on comprend facilement que si l'on enlève la logique combinatoire d'une machine à états finis, il reste uniquement un registre connecté à lui-même. L'effort jugé pour implémenter cette solution est modéré. En effet, il suffirait d'implémenter un algorithme qui parcourt le graphe contracté dans lequel uniquement les multiplexeurs de test et les registres sont représentés. Chaque fois que l'algorithme passerait par un registre, il vérifierait si celui-ci possède une connexion sur lui-même. On pourrait ensuite fixer un seuil du nombre de registres bouclant sur eux-mêmes à partir duquel il est nécessaire d'insérer un module de test de machine à états finis.

Du point de vue de la génération automatisé des vecteurs de tests, on pourrait couvrir les pannes de la structure du réseau d'horloges. Pour cela, il faudrait repenser les *key-units* du mini-MIPS en prenant en compte leurs tests. On parle ici de changer profondément l'architecture du mini-MIPS, l'effort exigé serait donc très important.

Enfin d'un point de vue pratique, la génération des procédures de test peut être facilitée, toute l'information nécessaire à son automatisation est présente dans le programme de connexion des signaux de contrôle de test. L'effet jugé à l'implémentation de cette automatisation est donc modéré.

Pour finir, on rappelle que pour connaître la direction des ports des cellules standard, l'algorithme se base sur la liste des ports considérés comme des sorties dans le fichier de configuration. Les ports qui ne sont pas dans cette liste sont interprétés comme des entrées. L'implémentation de cette technique a eu pour avantage de nous faire gagner beaucoup de temps de développement, mais la solution n'est pas très robuste. Il conviendrait plutôt de lire la librairie de la technologie utilisée pour en tirer les caractéristiques des ports des différentes

cellules. On juge l'effort nécessaire à cette implémentation assez importante, car il faudrait créer un algorithme capable de parcourir n'importe quelle librairie de cellule standard.

ANNEXE I

SCRIPT TESSENT D'INSERTION DES STRUCTURES DE TEST CONVENTIONNELLES ET GÉNÉRIQUES

```
set_drc_handling s1 warn
set_drc_handling s2 warn
set_drc_handling s5 warn
set include_existing_chains 1
set_context dft -scan -hier
read_cell_library libcomp.atpglib
read_verilog ../implementation/minispim_core_without_reg/synth/netlist/minispim_core_minispim_
core_without_reg_net.v
set_current_design minispim_core

//adding clocks and signals
set_system_mode setup

set MainClock_list [open "MainClock.txt" r]
set SecondaryClock_list [open "SecondaryClock.txt" r]

while {[gets $MainClock_list line]!=-1} {

    if {[llength $line]!=1} {
        puts "Incorrect format in the line -> '$line'"
        continue
    }
    foreach {mclock} $line {
        puts ->>>$mclock;
    }

    add_clocks 0 $mclock -internal

}

close $MainClock_list

while {[gets $SecondaryClock_list line]!=-1} {

    if {[llength $line]!=1} {
        puts "Incorrect format in the line -> '$line'"
        continue
    }
    foreach {sclock} $line {
        puts ->>>$sclock;
    }

    add_clocks 0 $sclock -internal

}


close $SecondaryClock_list

report_clocks
report_primary_inputs
set_scan_signals -TClk scan_clk -TEn scan_en
set_test_logic -set ON -reset ON -clock off -tristate off -ram off
set_scan_enable -single_global_scan_enable off
check_design_rules
```

```
// Insert scan
set_scan_insertion_options -port_index_start_value 1 -si_port_format {%s_scan_in%d} -
so_port_format {%s_scan_out%d} -si_timing any_edge -so_timing any_edge -
lockup_cell_type latch -si_lockup_cell_type latch -so_lockup_cell_type latch -
single_clock_edge_chains ON -single_clock_domain_chains ON -single_wrapper_type_chains ON
set_insertion_options -module_uniquification_suffix _scan#
set_scan_insertion_options -single_clock_domain_chains off
set_scan_insertion_options -single_clock_edge_chains off
set intest_families {}
set extest_families {}
  if { [llength $intest_families] } {
  add_scan_mode unwrapped -include_elements [get_scan_elements] -
si_port_format {%sscan_in%d} -so_port_format {%sscan_out%d} -port_index_start_value 1 -
port_scalar_index_modifier 1 -include_chain_families $intest_families
} else {
  add_scan_mode unwrapped -include_elements [get_scan_elements] -
si_port_format {%sscan_in%d} -so_port_format {%sscan_out%d} -port_index_start_value 1 -
port_scalar_index_modifier 1
}

analyze_scan_chains
insert_test_logic
report_scan_chains
report_scan_cells
set_system_mode insertion

create_port scan_clk -direction input
create_port scan_m0 -direction input

set MainClock_list [open "MainClock.txt" r]

while {[gets $MainClock_list line]!=-1} {

    if {[llength $line]!=1} {
        puts "Incorrect format in the line -> '$line'"
        continue
    }
    foreach {mclock} $line {
        puts ->>>$mclock;
    }

   #format string for intercept_connection
    set s [string map {. _} $mclock]
    set s [string map {[ _} $s]
    set s [string map {] _} $s]
    set s [string map {/ _} $s]

    intercept_connection $mclock -cell_function_name mux -input2 scan_clk -select scan_m0 -
leaf_instance ${s}_
}

close $MainClock_list
write_design -output_file minispim_scan_without_reg_withfinalscript.v -replace

write_atpg_setup test_atpg_setup -replace
set_system_mode setup
report_sequential_instances -
Format PN > minispim_scan_without_reg_register_withfinalscript.txt
```

ANNEXE II

SCRIPT TESSENT D'INSERTION DES MODULES DE TEST DE MACHINE À ÉTATS FINIS *FSM_TEST*

```
set_drc_handling s1 warn
set_drc_handling s2 warn
set_drc_handling s5 warn
set include_existing_chains 1
set_context dft -scan -hier
read_cell_library libcomp.atpglib
read_verilog netlist_with_bypassku.v
read_verilog ../implementation/FSM_test/synth/netlist/FSM_test.v

set_current_design minispim_core
set_system_mode insertion

create_port fsm_mode -direction input
create_port rstn_fsm_test -direction input

set Mux_FSM_list [open "FSMmodule.txt" r]
set FSM_test_number 0

while {[gets $Mux_FSM_list line]!=-1} {

    if {[llength $line]!=1} {
        puts "Incorrect format in the line -> '$line'"
        continue
    }
    foreach {DelayMux} $line {
        puts ->>>$DelayMux;
        incr FSM_test_number
        create_instance FSM_test$FSM_test_number -of_module FSM_test
        create_connections [get_fanins $DelayMux/A -stop_on cell_pin] FSM_test$FSM_test_number/clk_in
        create_connections [get_fanins $DelayMux/S0 -stop_on cell_pin] FSM_test$FSM_test_number/scan_mode
        create_connections [get_fanins $DelayMux/B -stop_on cell_pin] FSM_test$FSM_test_number/scan_clk
        move_connections -from $DelayMux/Y -to FSM_test$FSM_test_number/clk_out
        create_connections rstn_fsm_test FSM_test$FSM_test_number/reset_n
        create_connections fsm_mode FSM_test$FSM_test_number/fsm_mode
        delete_instances $DelayMux

    }

}
close $Mux_FSM_list

set_system_mode setup

write_design -output_file final_dft_minispim.v -replace
```

ANNEXE III

SCRIPT TESSENT D'INSERTION DES MODULES DE CONTOURNEMENT DES KEY UNITS *BYPASS_KU*

```
set_drc_handling s1 warn
set_drc_handling s2 warn
set_drc_handling s5 warn
set include_existing_chains 1
set_context dft -scan -hier
read_cell_library libcomp.atpglib
read_verilog ../implementation/TestModeOrClockConnect/synth/netlist/TestModeOrClockConnect_TestModeOr-
ClockConnect_net.v
read_verilog ../final_netlist/scan_netlist_out_without_reg.v
set_current_design minispim_core
set_system_mode insertion

create_port test_mode -direction input
create_port init_TestOrModule_input -direction input
create_port scan_mode_or -direction input

set TestModeOrmModuleNumber 0
set ClockInputNumber 0
set Clock_dependence_list [open "OrTestModeConnectionReport_without_reg.txt" r]

while {[gets $Clock_dependence_list line]!=-1} {

    if {[llength $line]!=2} {
        puts "Incorrect format in the line -> '$line'"
        continue
    }
    foreach {var1 var2 } $line {
        puts ->>>$var1;puts ->>>$var2;
    }

    if { $var1 == ".delay_input" } {
        set DelayInput $var2
        incr TestModeOrmModuleNumber
        create_instance TestModuleOR$TestModeOrmModuleNumber -of_module TestModeOrClockConnect
        set ClockInputNumber 0
        set OldDelayInputConnection [get_fanins $DelayInput/A -stop_on cell_pin]
        set my_pins [get_fanouts $OldDelayInputConnection]
        puts "my_pins are [get_name_list $my_pins]"
        foreach_in_collection pin $my_pins {
            delete_connections $pin
            create_connections TestModuleOR$TestModeOrmModuleNumber/clock_output $pin
        }
        create_connections $OldDelayInputConnection TestModuleOR$TestModeOrmModuleNumber/in_clock
        create_connections test_mode TestModuleOR$TestModeOrmModuleNumber/test_mode
        create_connections scan_mode_or TestModuleOR$TestModeOrmModuleNumber/scan_mode
        create_connections scan_clk TestModuleOR$TestModeOrmModuleNumber/scan_clk

    } elseif { $var1 == ".clock_depend" } {
        if { $var2 == "0" } {
            create_connections init_TestOrModule_input TestModuleOR$TestModeOrmModuleNumber/clock_$ClockIn-
putNumber
        } else {
            create_connections $var2/Y TestModuleOR$TestModeOrmModuleNumber/clock_$ClockInputNumber
        }
        incr ClockInputNumber
    } else {
        puts "Error, wrong command"
    }
}
close $Clock_dependence_list
write_design -output_file netlist_with_bypassku.v -replace
```

ANNEXE IV

SCRIPT GENUS POUR APLATIR LA NETLIST ET EXTRAIRE LES DÉLAIS

```
set VERBOSE_LEVEL  7;           # 0-11

source $::env(SCRIPTDIR)/globals.tcl

set src_verilog ""
lappend src_verilog "$::env(FINAL_NETDIR)/final_dft_minispim.v"

# Information level
set_db information_level $VERBOSE_LEVEL

# Libraries
set_db init_lib_search_path [list $::env(LIB_SC_TIMING) \
                                  $::env(LIB_SC_LEF)    \
                                  $::env(LIB_IO_LEF) \
                                      $::env(LIB_SC_QRC)
                                ]

set_db init_hdl_search_path [list $::env(HDLDIR)]

#Exclude scan flip-flop from mapping
create_library_domain {scan_dom non_scan_dom}

set_db library_domain:scan_dom .library {fast_vdd1v0_basicCells.lib fast_vdd1v0_extvdd1v0.lib
fast_vdd1v0_multibitsDFF.lib}

set_db library_domain:non_scan_dom .library {fast_vdd1v0_basicCells.lib fast_vdd1v0_extvdd1v0.
lib fast_vdd1v0_multibitsDFF.lib}
get_db lib_cells -regexp SDFFR* -foreach {set_db $object .avoid true}
set_db lef_library {gsclib045_tech.lef gsclib045_macro.lef gsclib045_multibitsDFF.lef}

set_db qrc_tech_file $LIB_SCQRC

# Turn on vhdl 2008 support
set_db hdl_vhdl_read_version 2008

read_hdl -v2001 $src_verilog

elaborate $::env(DESIGN)

# Check unresolved references
check_design -unresolved

ungroup -all -simple
uniquify $::env(DESIGN)

# Final netlist
write_hdl > $::env(FINAL_NETDIR)/final_netlist_w_dft_a_sdf.v
write_sdf > $::env(FINAL_NETDIR)/final_netlist_w_dft_a_sdf.sdf
```

ANNEXE V

SCRIPT TESSENT DE GÉNÉRATION AUTOMATISÉE DES VECTEURS DE TEST

```
set_context patterns -scan
read_verilog final_netlist_w_dft_a_sdf_without_reg.v
read_cell_library libcomp.atpglib
set_current_design minispim_core
create_flat_model
read_sdf final_netlist_w_dft_a_sdf_without_reg.sdf -maximum_delay

add_scan_groups grp1 ./test_atpg_setup_minimips_ext_sep_without_reg_converg.testproc
add_scan_chains chain1 grp1 scan_in1 scan_out1

add_clocks 0 \gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[0].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[0].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[0].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y -internal

add_clocks 0 \gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[1].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[1].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y -internal

add_clocks 0 \gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[2].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[2].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y -internal
add_clocks 0 \gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y -internal

add_clocks 0 scan_clk
add_input_constraints -c0 init_TestOrModule_input

set fault type transition -NO_Shift_launch
set abort limit 500 500
set_pattern_type -sequential 2
set_clock_restriction off

set system mode analysis

set mode internal
set_capture_procedures on -All

create_patterns

//write_patterns mips_patterns.v -Verilog -replace -
PROcfile ./test_atpg_setup_minimips.testproc -VERBose -MODE_Internal -
PARAMeter_file ./gen_vectors.param
write_patterns pat_serial.v -verilog -replace -Serial -MODE_External -SAMple
write_faults undetected_mips.txt -replace -class  AU
write_faults DImips.txt -replace -class  DI
write_faults DSmips.txt -replace -class  DS
write_faults uncontrolled_mips.txt -replace -class UC
```

ANNEXE VI

SCRIPT TESSENT DE PROCÉDURE DE TEST DU MINI-MIPS POUR LE *LAUNCH-ON-CAPTURE* PARTIELLEMENT À VITESSE NOMINALE NÉCESSAIRE À LA GÉNÉRATION AUTOMATISÉE DES VECTEURS DE TEST

```
alias alias_allclock = "\gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y" ,
"\gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";

alias alias_clock0 = "\gen_xu[0].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_W_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1 = "\gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3 ="\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";

alias alias_clock0_1 = "\gen_xu[1].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_I_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_1 = "\gen_xu[1].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_P_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2_1 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_1 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";

alias alias_clock0_2 = "\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y",
"\gen_xu[2].u_eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_2 = "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock2_2 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_2 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";

alias alias_clock0_3 = "\gen_xu[0].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y","\gen_xu[0].u_
eu_u_dl_R_GEN_DELAI[26].Delai_output.DO/Y","\gen_xu[1].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.
DO/Y","\gen_xu[2].u_eu_u_dl_D_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock1_3 = "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
```

```
alias alias_clock2_3 = "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
alias alias_clock3_3 = "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";

set time scale 0.1000000 ns ;
 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 100 ;
    pulse scan_clk 200 20;
    period 400 ;
 end;

 procedure test_setup =
    timeplate gen_tp1 ;
    // cycle 0 starts at time 0
    cycle =
        force rst_n 1;
        force rstn_fsm_test 0;
        force scan_en 0 ;
        force scan_reset 0;
        force scan_mode_0 1 ;
        force fsm_mode 0;
        force scan_mode_1 0;
        force scan_mode_2 0 ;
        force scan_mode_3 0;
        force scan_mode_1A 0;
        force scan_mode_or 0;
        force end_graph 0;
        force test_mode 1;
        force init_TestOrModule_input 0;
    end;
    cycle =
        force rst_n 1;
        force rstn_fsm_test 0;
        force scan_en 0;
        force scan_reset 0;
        force fsm_mode 0;
        force scan_mode_0 0;
        force scan_mode_1 1;
        force scan_mode_2 0 ;
        force scan_mode_3 0;
        force scan_mode_1A 0;
        force scan_mode_or 0;
        force end_graph 0;
        force test_mode 1;
    end;
    cycle =
        force rst_n 1;
        force rstn_fsm_test 0;
        force scan_en 0;
        force scan_reset 0;
        force fsm_mode 0;
        force scan_mode_0 0;
        force scan_mode_1 0;
        force scan_mode_2 1 ;
        force scan_mode_3 0;
        force scan_mode_1A 0;
        force end_graph 0;
        force test_mode 1;
        force scan_mode_or 0;
    end;
    cycle =
```

```
      force rst_n 1;
      force rstn_fsm_test 0;
      force scan_en 0;
      force scan_reset 0;
      force fsm_mode 0;
      force scan_mode_0 0;
      force scan_mode_1 0;
      force scan_mode_2 0 ;
      force scan_mode_3 1;
      force scan_mode_1A 0;
      force end_graph 0;
      force test_mode 1;
      force scan_mode_or 0;
   end;
   cycle =
      force rst_n 0;
      force rstn_fsm_test 0;
      force scan_en 0;
      force scan_reset 0;
      force fsm_mode 0;
      force scan_mode_0 1;
      force scan_mode_1 1;
      force scan_mode_2 1 ;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force end_graph 1;
      force test_mode 1;
      force scan_mode_or 0;
   end;
   cycle =
      force rst_n 1;
      force rstn_fsm_test 0;
      force scan_en 0;
      force scan_reset 1;
      force fsm_mode 0;
      force scan_mode_0 1;
      force scan_mode_1 1;
      force scan_mode_2 1 ;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force end_graph 1;
      force test_mode 1;
      force scan_mode_or 0;
   end;
end;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   // cycle 0 starts at time 0
   cycle =
      force_sci ;
      measure_sco ;
      pulse scan_clk ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   // cycle 0 starts at time 0
```

```
    cycle =
        force scan_reset 0;
        force scan_set 0;
        force alias_allclock 0;
        force scan_clk 0;
        force scan_en 1;
        force scan_mode_0 1;
        force scan_mode_1 1;
        force scan_mode_2 1;
        force scan_mode_3 1;
        force scan_mode_1A 1;
        force end_graph 1;
        force scan_mode_or 1;
        force fsm_mode 0;
        force rst_n 1;
        force rstn_fsm_test 1;
        force test_mode 1;
        force init_TestOrModule_input 0;
    end ;
    apply shift 978;
 end;

// Launch timeplate
//-------------------
timeplate tp_launch_ext =
  force_pi 0 ;
  measure_po 0 ;
  pulse scan_clk 50 20 ;
  period 70 ;
end ;

// Capture  timeplate
//-------------------
timeplate tp_capture_all =
  force_pi 0 ;
  measure_po 5 ;
  pulse scan_clk 22 20;
  period 50 ;
end ;

timeplate tp_capture_123 =
  force_pi 0 ;
  measure_po 0 ;
  pulse alias_clock1 22 20;
  pulse alias_clock2 22 20;
  pulse alias_clock3 22 20;
  period 50 ;
end ;

timeplate tp_capture_023 =
  force_pi 0;
  measure_po 0;
  pulse alias_clock0_1 22 20;
  pulse alias_clock2_1 22 20;
  pulse alias_clock3_1 22 20;
  period 50 ;
end ;

timeplate tp_capture_013 =
  force_pi 0;
  measure_po 5;
```

```
  pulse alias_clock0_2 22 20;
  pulse alias_clock1_2 22 20;
  pulse alias_clock3_2 22 20;
  period 50 ;
end ;

timeplate tp_capture_012 =
  force_pi 0 ;
  measure_po 5;
  pulse alias_clock0_3 22 20;
  pulse alias_clock1_3 22 20;
  pulse alias_clock2_3 22 20;
  period 50 ;
end ;

timeplate tp_capture_A2 =
  force_pi 0;
  measure_po 5;
  pulse "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y" 22 20;
  period 50 ;
end ;

timeplate tp_capture_A1 =
  force_pi 0;
  measure_po 5;
  pulse "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y" 22 20 ;
  period 50 ;
end ;

timeplate tp_capture_A0 =
  force_pi 0;
  measure_po 5;
  pulse "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y" 22 20 ;
  period 50 ;
end ;

timeplate prep_test =
  force test_mode 1;
  force init_TestOrModule_input 0;
  period 100;
end;

procedure capture procedure_at_speed_1 =
  // Internal mode
  //----------------------------------------
  mode internal =
     // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 0 ;
      force scan_mode_2 0;
      force scan_mode_3 0;
      force scan_mode_1A 0;
      force end_graph 0;
      force scan_mode_or 0;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
```

```
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
      //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_123 ;
      measure_po;
      pulse alias_clock1   ;
      pulse alias_clock2;
      pulse alias_clock3;
    end;
  end;

  // External mode
  // ( replicates of the internal mode without internal clocks )
  //----------------------------------------
  mode external =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 0 ;
      force scan_mode_2 0;
      force scan_mode_3 0;
      force scan_mode_1A 0;
      force end_graph   0;
      force scan_mode_or 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end ;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_123;
      measure_po;
    end;
  end ;
end;


procedure capture procedure_at_speed_2 =
  mode internal =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 0;
      force scan_mode_1 1 ;
      force scan_mode_2 0;
```

```
      force scan_mode_3 0;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 0;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_023 ;
      measure_po;
      pulse alias_clock0_1;
      pulse alias_clock2_1;
      pulse alias_clock3_1;
    end;
  end;
  mode external =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 0;
      force scan_mode_1 1 ;
      force scan_mode_2 0;
      force scan_mode_3 0;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph   0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end ;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_023;
      measure_po;
    end;
  end;
end;

procedure capture procedure_at_speed_3 =
  mode internal =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
```

```
      force scan_mode_0 0;
      force scan_mode_1 0 ;
      force scan_mode_2 1;
      force scan_mode_3 0;
      force scan_mode_1A 0;
      force scan_mode_or 0;
      force end_graph 0;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_013;
      measure_po;
      pulse alias_clock0_2;
      pulse alias_clock1_2;
      pulse alias_clock3_2;
    end;
  end;
  mode external =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 0;
      force scan_mode_1 0 ;
      force scan_mode_2 1;
      force scan_mode_3 0;
      force scan_mode_1A 0;
      force scan_mode_or 0;
      force end_graph   0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end ;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_013;
      measure_po;
    end;
  end;
end;

procedure capture procedure_at_speed_4 =
  mode internal =
    // Cycle 1 : launch pulses
```

```
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 0;
      force scan_mode_1 0 ;
      force scan_mode_2 0;
      force scan_mode_3 1;
      force scan_mode_1A 0;
      force scan_mode_or 0;
      force end_graph 0;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_012;
      measure_po;
      pulse alias_clock0_3;
      pulse alias_clock1_3;
      pulse alias_clock2_3;
    end;
  end;
  mode external =
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 0;
      force scan_mode_1 0 ;
      force scan_mode_2 0;
      force scan_mode_3 1;
      force scan_mode_1A 0;
      force end_graph   0;
      force scan_mode_or 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end ;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_012;
      measure_po;
    end;
  end;
end;
```

```
procedure capture procedure_at_speed_FSM_A2 =
  mode internal =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force end_graph 1;
      force test_mode 1;
      force scan_mode_or 0;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
    end;
  // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 0;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
  //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A2;
      measure_po;
      pulse "\gen_xu[2].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
    end;
  end;
  mode external =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
```

```
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
    end;
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1;
      force scan_mode_2 1;
      force scan_mode_3 0;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph   1;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A2;
      measure_po;
    end;
  end;
end;


procedure capture procedure_at_speed_FSM_A1 =
  mode internal =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
```

```
      force rst_n 1;
      force scan_clk  0 ;
    end;
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 0;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A1;
      measure_po;
      pulse "\gen_xu[1].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
    end;
  end;
  mode external =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
    end;
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1;
      force scan_mode_2 0;
      force scan_mode_3 1;
```

```
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph   1;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A1;
      measure_po;
    end;
  end;
end;

procedure capture procedure_at_speed_FSM_A0 =
  mode internal =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
    end;
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 0;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
```

```
force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end ;
    //  // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A0;
      measure_po;
      pulse "\gen_xu[0].u_eu_u_dl_A_GEN_DELAI[26].Delai_output.DO/Y";
    end;
  end;
  mode external =
    cycle =
      timeplate prep_test;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 0;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 0;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
    end;
    // Cycle 1 : launch pulses
    cycle =
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 0;
      force scan_mode_or 0;
      force end_graph   1;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
      force test_mode 1;
      force fsm_mode 1;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
    end;
    // Cycle 2 : capture pulses
    cycle  =
      timeplate tp_capture_A0;
      measure_po;
    end;
  end;
```

```
end;

procedure capture procedure_at_speed_all =
  mode internal =
    cycle=
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 1;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end;
    cycle=
      timeplate tp_capture_all;
      pulse scan_clk;
      measure_po;
    end;
  end;
  mode external =
    cycle=
      timeplate tp_launch_ext ;
      force_pi ;
      force scan_mode_0 1;
      force scan_mode_1 1 ;
      force scan_mode_2 1;
      force scan_mode_3 1;
      force scan_mode_1A 1;
      force scan_mode_or 1;
      force end_graph 1;
      force test_mode 1;
      force fsm_mode 0;
      force rstn_fsm_test 1;
      force init_TestOrModule_input 0;
      force scan_reset 0;
      force scan_set 0;
      force scan_en 0;
      force rst_n 1;
      force scan_clk  0 ;
      pulse scan_clk ;
    end;
    cycle=
      timeplate tp_capture_all;
      pulse scan_clk;
      measure_po;
    end;
  end;
end;
```

ANNEXE VII

RAPPORT *BYPASS_KU* DU MINI-MIPS

```
.delay_input gen_xu[0].u_eu_u_dl_A_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_1_D_mux
.clock_depend delay_2_W_mux
.clock_depend delay_1_R_mux
.clock_depend delay_0_W_mux
.clock_depend delay_0_D_mux
.clock_depend delay_1_A_mux
.clock_depend delay_0_R_mux
.clock_depend delay_2_R_mux
.clock_depend delay_2_D_mux
.clock_depend delay_2_A_mux
.clock_depend delay_1_W_mux
.delay_input gen_xu[1].u_eu_u_dl_P_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_1_W_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[0].u_eu_u_dl_W_GEN_DELAI[0].Delai_input.DI
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[1].u_eu_u_dl_R_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_A_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[0].u_eu_u_dl_P_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_W_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[0].u_eu_u_dl_I_GEN_DELAI[0].Delai_input.DI
```

```
.clock_depend delay_0_P_mux
.clock_depend delay_1_P_mux
.clock_depend delay_2_P_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[0].u_eu_u_dl_D_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_A_mux
.clock_depend delay_1_A_mux
.clock_depend delay_2_A_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[0].u_eu_u_dl_R_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_2_A_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[1].u_eu_u_dl_A_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_1_D_mux
.clock_depend delay_0_A_mux
.clock_depend delay_2_W_mux
.clock_depend delay_1_R_mux
.clock_depend delay_0_W_mux
.clock_depend delay_0_D_mux
.clock_depend delay_0_R_mux
.clock_depend delay_2_R_mux
.clock_depend delay_2_D_mux
.clock_depend delay_2_A_mux
.clock_depend delay_1_W_mux
.delay_input gen_xu[1].u_eu_u_dl_D_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_A_mux
.clock_depend delay_1_A_mux
.clock_depend delay_2_A_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[1].u_eu_u_dl_I_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_P_mux
```

```
.clock_depend delay_1_P_mux
.clock_depend delay_2_P_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[2].u_eu_u_dl_D_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_A_mux
.clock_depend delay_1_A_mux
.clock_depend delay_2_A_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[2].u_eu_u_dl_W_GEN_DELAI[0].Delai_input.DI
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[1].u_eu_u_dl_W_GEN_DELAI[0].Delai_input.DI
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[2].u_eu_u_dl_A_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_1_D_mux
.clock_depend delay_0_A_mux
.clock_depend delay_2_W_mux
.clock_depend delay_1_R_mux
.clock_depend delay_0_W_mux
.clock_depend delay_0_D_mux
.clock_depend delay_1_A_mux
.clock_depend delay_0_R_mux
.clock_depend delay_2_R_mux
.clock_depend delay_2_D_mux
.clock_depend delay_1_W_mux
.delay_input gen_xu[2].u_eu_u_dl_R_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_1_A_mux
.clock_depend 0
```

```
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[2].u_eu_u_dl_I_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_0_P_mux
.clock_depend delay_1_P_mux
.clock_depend delay_2_P_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.delay_input gen_xu[2].u_eu_u_dl_P_GEN_DELAI[0].Delai_input.DI
.clock_depend delay_2_W_mux
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
.clock_depend 0
```

ANNEXE VIII

RAPPORT *FSM_TEST* DU MINI-MIPS

```
delay_0_A_mux
delay_1_A_mux
delay_2_A_mux
```

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Beest, F. T., Peeters, A., Verra, M., Berkel, K. v., & Kerkhoff, H. (2002). Automatic scan insertion and test generation for asynchronous circuits. Dans *Proceedings. International Test Conference* (pp. 804-813). doi: 10.1109/TEST.2002.1041834

Belete, D., Razdan, A., Schwarz, W., Raina, R., Hawkins, C., & Morehead, J. (2002). Use of DFT techniques in speed grading a 1 GHz+ microprocessor. Dans *Proceedings. International Test Conference* (pp. 1111-1119). IEEE.

Benware, B., Lu, C., Slyke, J. V., Prabhu, K., Madge, R., Keim, M., . . . Rajski, J. (2004). Affordable and effective screening of delay defects in ASICs using the inline resistance fault model. Dans *2004 International Conferce on Test* (pp. 1285-1294). doi: 10.1109/TEST.2004.1387403

Berthier, F. (2016). *Conception d'un processeur ultra basse consommation pour les noeuds de capteurs sans fil* (Thèse, Université Rennes 1).

Brégier, V. (2007). *Automatic synthesis of optimised proven quasi delay insensitive asynchronous circuits* (Thèse, Institut National Polytechnique de Grenoble - INPG). Repéré à https://tel.archives-ouvertes.fr/tel-00178543

Cogis, O., & Robert, C. (2003). *Théorie des graphes: au-delà des ponts de Königsberg: problèmes, théorèmes, algorithmes*. Vuibert.

Corno, F., Reorda, M. S., & Squillero, G. (2000). RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers, 17*(3), 44-53. doi: 10.1109/54.867894

David, I., Ginosar, R., & Yoeli, M. (1995). Self-timed is self-checking. *Journal of Electronic Testing, 6*(2), 219-228.

Fiorentino, M. (2018). KeyRing − A Self-Timed Design Template Compatible With Conventional EDA Flows

Gill, G., Agiwal, A., Singh, M., Feng, S., & Makris, Y. (2006). Low-overhead testing of delay faults in high-speed asynchronous pipelines. Dans *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)* (pp. 11 pp.-56). doi: 10.1109/ASYNC.2006.20

HandshakeSolutions. (2004). Handshake Solutions HT80C51. Repéré à http://www.keil.com/dd/chip/3931.htm

Hasib, O. A.-T., Crépeau, D., Awad, T., Dulipovici, A., Savaria, Y., & Thibeault, C. (2018). Exploiting built-in delay lines for applying *launch-on-capture* at-speed testing on self-timed circuits. Dans *VLSI Test Symposium (VTS), 2018 IEEE 36th* (pp. 1-6). IEEE.

Hodson, H. (2015). The battery revolution that will let us all be power brokers. *New Scientist*.

Hulgaard, H., Burns, S. M., & Borriello, G. (1995). Testing asynchronous circuits: A survey. *Integration, the VLSI journal, 19*(3), 111-131.

John Ellson, E. G., Lefteris Koutsofios, Stephen North, Gordon Woodhull, Lucent Technologies. (2001). Graphviz — open source graph drawing tools. Dans *Lecture Notes in Computer Science* (pp. 483--484). Springer-Verlag.

Keutzer, K., Lavagno, L., & Sangiovanni-Vincentelli, A. (1991). Synthesis for testability techniques for asynchronous circuits. Dans *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers* (pp. 326-329). IEEE.

Kishinevsky, M., Kondratyev, A., Lavagno, L., Saldanha, A., & Taubin, A. (1998). Partial-scan delay fault testing of asynchronous circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17*(11), 1184-1199. doi: 10.1109/43.736191

Kondratyev, A., & Lwin, K. (2002). Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers, 19*(4), 107-117. doi: 10.1109/MDT.2002.1018139

Kondratyev, A., Sorensen, L., & Streich, A. (2002). Testing of asynchronous designs by "inappropriate" means. Synchronous approach. Dans *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems* (pp. 171-180). doi: 10.1109/ASYNC.2002.1000307

Laurence, M. (2012). Introduction to Octasic asynchronous processor technology. Dans *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems* (pp. 113-117). IEEE.

Lavagno, L., Lioy, A., & Kishinevsky, M. (1994). Testing redundant asynchronous circuits by variable phase splitting. Dans *Proceedings of the conference on European design automation* (pp. 328-333). IEEE Computer Society Press.

Leskovec, J. a. S., Rok. (2016). SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST), 8*, 1.

Marc Renaudin, P. V., Alex Yakovlev. (2013). Advances in Asynchronous logic from Principles to GALS & NoC,Recent Industry Applications, and Commercial CAD tools. 10.

Maxwell, P., Hartanto, I., & Bentz, L. (2000). Comparing functional and structural tests. Dans *Test Conference, 2000. Proceedings. International* (pp. 400-407). IEEE.

Mickaël Fiorentino, Y. S., Claude Thibeault. (2017). *Exploration Architecturale de Microprocesseurs Token-Based Self-Timed*. (Projet de thèse, Polytechnique Montréal)

Mickaël Fiorentino, Y. S., Claude Thibeault. (2017). FPGA Implementation of Token-Based Self-Timed Processors: A Case Study. 4.

Park, I., & McCluskey, E. J. (2008). Launch-on-Shift-Capture Transition Tests. Dans *2008 IEEE International Test Conference* (pp. 1-9). doi: 10.1109/TEST.2008.4700648

Petlin, O. A., & Furber, S. B. (1995). Scan testing of micropipelines. Dans *VLSI Test Symposium, 1995. Proceedings., 13th IEEE* (pp. 296-301). IEEE.

Piestrak, S. J., & Nanya, T. (1995). Towards totally self-checking delay-insensitive systems. Dans *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on* (pp. 228-237). IEEE.

Rios, D. (2008). *Systeme a microprocesseur asynchrone basse consommation* (Thèse, Institut National Polytechnique de Grenoble-INPG).

Satagopan, V., Bhaskaran, B., Al-Assadi, W., & Smith, S. C. (2005). Automation in design for test for asynchronous null conventional logic (NCL) circuits. Dans *12th NASA Symp. VLSI Des., Coeur d'Alene, ID*.

Satagopan, V., Bhaskaran, B., Al-Assadi, W. K., Smith, S. C., & Kakarla, S. (2007). DFT Techniques and Automation for Asynchronous NULL Conventional Logic Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15*(10), 1155-1159. doi: 10.1109/TVLSI.2007.903945

Savir, J., & Patil, S. (1993). Scan-based transition test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12*(8), 1232-1241. doi: 10.1109/43.238615

Savir, J., & Patil, S. (1994). Broad-side delay test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13*(8), 1057-1064.

Shi, F., & Makris, Y. (2006). Testing Delay Faults in Asynchronous Handshake Circuits. Dans *2006 IEEE/ACM International Conference on Computer Aided Design* (pp. 193-197). doi: 10.1109/ICCAD.2006.320085

Shinde, J., & Salankar, S. S. (2011). Clock gating — A power optimizing technique for VLSI circuits. Dans *2011 Annual IEEE India Conference* (pp. 1-4). doi: 10.1109/INDCON.2011.6139440

Smith, G. L. (1985). Model for Delay Faults Based Upon Paths. Dans *ITC* (pp. 342-351). Citeseer.

Spars, J., & Furber, S. (2002). *Principles asynchronous circuit design*. Springer.

Sparsø, J. (2001). Asynchronous circuit design-a tutorial. Dans *Chapters 1-8 in" Principles of asynchronous circuit design-A systems Perspective"*. Kluwer Academic Publishers.

te Beest, F., & Peeters, A. (2005). A multiplexer based test method for self-timed circuits. Dans *11th IEEE International Symposium on Asynchronous Circuits and Systems* (pp. 166-175). IEEE.

Te Beest, F., Peeters, A., Van Berkel, K., & Kerkhoff, H. (2003). Synchronous full-scan for asynchronous handshake circuits. *Journal of Electronic Testing, 19*(4), 397-406.

TÊTU, J.-F. (2014). *Caractérisation et analyse des chemins critiques de circuits intégrés asynchrones complexes* (Mémoire, Université du Québec).

Touati, A. (2016). *Improving Functional and Structural Test Solutions for Integrated Circuits* (Thèse, Université Montpellier).

van Berkel, K., Peeters, A., & te Beest, F. (2003). Adding synchronous and LSSD modes to asynchronous circuits. *Microprocessors and Microsystems, 27*(9), 461-471.

Van Gageldonk, H., Van Berkel, K., Peeters, A., Baumann, D., Gloor, D., & Stegmann, G. (1998). An asynchronous low-power 80c51 microcontroller. Dans *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on* (pp. 96-107). IEEE.

Verma, K., Kaushik, B., & Singh, R. (2010). Propagation delay variations under process deviation in driver interconnect load system. Dans *2010 International Conference on Advances in Recent Technologies in Communication and Computing* (pp. 408-410). IEEE.

Waicukauski, J. A., Lindbloom, E., Rosen, B. K., & Iyengar, V. S. (1987). Transition fault simulation. *IEEE Design & Test of Computers, 4*(2), 32-38.

Wang, L.-T., Wu, C.-W., & Wen, X. (2006). *VLSI test principles and architectures: design for testability*. Elsevier.

Wolf, C. (2012). Yosys Open Synthesis Suite. Repéré à http://www.clifford.at/yosys/about.html

Zeidler, S., & Krstić, M. (2015). A survey about testing asynchronous circuits. Dans *2015 European Conference on Circuit Theory and Design (ECCTD)* (pp. 1-4). doi: 10.1109/ECCTD.2015.7300128

Zeng, J., & Abadir, M. (2004). On correlating structural tests with functional tests for speed binning. Dans *Proceedings. 2004 IEEE International Workshop on Current and Defect Based Testing (IEEE Cat. No. 04EX1004)* (pp. 79-83). IEEE.