Synthèse de haut niveau pour la testabilité en-ligne

M.A. Naal

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA**
dans le cadre de **l'Ecole Doctorale « Electronique, Electrotechnique, Automatique, Télécommunications, Signal »**

présentée et soutenue publiquement

par

Mouhamad Ayman NAAL

le **24 septembre 2002**

Titre :

Synthèse de haut niveau pour la testabilité en-ligne

Directeurs de thèse :

Michael **NICOLAIDIS**
Emmanuel **SIMEU**

JURY

M. **R. LEVEUGLE,**	Président
M. **A. DANDACHE,**	Rapporteur
M. **S. J. PIESTRAK,**	Rapporteur
M. **M. NICOLAIDIS,**	Directeur de thèse
M. **E. SIMEU,**	Co-Directeur
M. **S. MIR,**	Examinateur

A ma mère et mon père qui ont attendu ce moment avec impatience
A mon épouse pour sa générosité, son support et sa patience
A Fatima et Zaynab

Remerciments

A la fin de cette expérience riche au laboratoire TIMA (Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs), je tiens à remercier M^r Bernard COURTOIS, le directeur du laboratoire, de m'avoir accueilli tout au longue de mes études.

Je remercie également M^r Michael NICOLAIDIS, le directeur de ma thèse. Ses larges expériences dans le domaine de la microélectronique m'ont aidé efficacement pour la définition du sujet de cette thèse.

Ma reconnaissance et mes remerciments sont dus à M^r Emmanuel SIMEU, le co-directeur de ma thèse. Je le remercie pour son suivi, pour les discussions que nous avons eu ensemble et pour son support et ses orientations.

Je ne saurais pas dissocier de ces remerciments le personnel du TIMA et celui du CIME. Plus particulièrement, je remercie M^r Alexandre CHAGOYA, le responsable du service conception au CIME, pour son aide et ses conseils.

Ma gratitude est bien due à tous les amis et collègues que j'ai connus. Ceux qui m'ont soutenu dans cette période d'étude. Ceux qui m'apportaient leurs conseils et leurs supports dans les moments difficiles. La mémoire de ces personnes restera pour moi une incitation au bienfait dans cette vie.

Avant-propos

La percée technologique dans la réalisation de circuits intégrés et les annonces récentes de l'apparition de transistors plus rapides et plus petits, permettra de nouvelles applications puissantes telles que l'identification en temps réel de voix et de visage, calculateurs sans claviers .... Côté fréquence, il est estimé que, dans quelques années, elle atteindra 10 GHz.

Pour ce faire, de nouvelles notions ont été avancées. On note particulièrement la notion de SoC (System-on-Chip) et les applications portables ou celles qui nécessitent une longue durée de vie avec une grande sûreté de fonctionnement comme dans les applications médicales, spatiales ou celles du transport où la fiabilité du système a son impact direct sur la vie de l'être humain et sur le succès de la mission. Ce problème de fiabilité ajoute une composante importante à la complexité croissante de systèmes numériques rendant ainsi la synthèse de tels systèmes une tâche difficile. De nouvelles solutions de synthèse de systèmes numériques sont donc nécessaires pour répondre à deux exigences : la manipulation de systèmes très complexes et l'intégration de méthodes de vérification en-ligne à haut niveau de synthèse.

Côté test en-ligne, cet avancement technologique permet aux systèmes numériques de disposer d'un intervalle oisif plus important. Cet intervalle est très utile pour l'application du test en-ligne, qu'il soit concurrent, semi-concurrent ou non-concurrent.

Cette étude propose une nouvelle méthode de synthèse de haut niveau qui tient compte des contraintes de test en-ligne. Elle est constituée essentiellement de deux parties. La première partie propose des nouvelles méthodes de test en-ligne non-concurrent et semi-concurrent. La deuxième partie propose une nouvelle approche de synthèse de haut niveau pour la testabilité en-ligne HLS_OLT (High Level Synthesis for On-Line Testability). Les deux parties sont assemblées pour fournir une solution intégrée de HLS_OLT.

Bien que cette solution améliore aussi bien la testabilité hors-ligne que la testabilité en-ligne, l'accent est mis sur la testabilité en-ligne des systèmes numériques. Outre le fait de l'amélioration de la testabilité, cette solution peut aussi permettre d'apporter une amélioration considérable en performances.

Deux notions, qui relèvent de l'intelligence artificielle, sont introduites. Premièrement, l'outil développé utilise un mécanisme d'auto-apprentissage. Chaque nouvelle expérience en synthèse de haut niveau est sauvegardée dans une base de données et sert comme guide pour les synthèses futures. Deuxièmement, les tâches de compilation et d'ordonnancement sont résolues par l'utilisation d'algorithmes génétiques. Ces deux techniques permettent le développement d'un système expert dont les performances s'améliorent avec chaque nouvelle expérience.

Table des Matières

Partie I

Généralités

Chapitre 1

Introduction

Le développement rapide de la technologie de fabrication de circuits et systèmes intégrés ainsi que la grande diversité des applications des systèmes numériques, ont imposé la recherche de nouvelles idées dans le domaine de l'architecture et l'organisation des ordinateurs ainsi que dans le domaine de la théorie du calcul numérique.

Pendant toute la durée de sa vie, de sa conception jusqu'à son fonctionnement au sein d'une application, un système intégré est soumis à différents tests ayant chacun des caractéristiques propres qui dépendent essentiellement des objectifs fixés, de la méthode appliquée et des outils utilisés.

Depuis la phase de conception, la conformité du circuit conçu par rapport aux spécifications initiales doit être garantie. Pour cela, le concepteur dispose d'un ensemple d'informations détaillées de diverses natures (topologique, électriques, logiques, ...) et des outils spécifiques (simulateur comportemental, logique et électrique) qui lui permettent une étude fine du comportement d'un élément quelconque du circuit.

Lors de la fabrication, le test consiste à vérifier que le circuit ne présente pas de défauts dûs au processus mis en œuvre dans les étapes technologiques de fabrication et qu'il est apte à fonctionner dans les plages prévues par chacun de ces paramètres (consommation, fréquence, ...). Le fabricant dispose pour cela de microscope à balayage électronique, de modèles de simulation, ...

Pendant son fonctionnement normal dans une application, le circuit doit être testé périodiquement pour déceler d'éventuelles défaillances. Les systèmes numériques sont, aujourd'hui, utilisés massivement dans des applications critiques. C'est le cas, par exemple, des applications médicales ou celles de transport où une défaillance a un impact direct sur la vie de l'être humain, ou le cas de systèmes ayant une mission dans des environnements critiques tels que les applications spatiales, sous terraines ou sous-marines où la défaillance peut entraîner des conséquences graves en sécurité et au niveau économique.

Ce type de systèmes doit garantir une très haute fiabilité et doit pouvoir réagir en cas de panne dans l'un de ses composants. Dans ce cas, la réaction du système commence par la détection en-ligne de cette panne le plus tôt possible et passe par la localisation du composant fautif pour terminer par une auto-configuration qui permettra au système de continuer son fonctionnement normal.

Le test en-ligne permet d'assurer la *readiness* du système pour une mission critique. Il consiste à vérifier le bon fonctionnement du système sans affecter son opération normale.

Pour ce faire, quatre types de redondances peuvent être exploités dans les composants du système. Ce sont les redondances en information, en temps, en matériel et en logiciel.

Le besoin de solutions de test en-ligne intégré est de plus en plus important. Malgré la complexité croissante de systèmes numériques, ces solutions doivent garantir un surcoût raisonnable en temps de conception, en ressources impliquées et en performance. Cela nécessite le développement d'une nouvelle méthode de synthèse de haut niveau qui doit garantir deux aspects. Le premier aspect est la possibilité de traiter des systèmes complexes à coût raisonnable. Le deuxième aspect est la prise en compte de la testabilité en-ligne dans les premières tâches du flot de la synthèse de haut niveau.

Pour s'accommoder à ce besoin, la présente étude propose deux axes de travail. Le premier axe consiste à proposer deux méthodes de test en-ligne, non-concurrent et semi-concurrent, présentées comme solutions intégrées. Le deuxième axe consiste à proposer une nouvelle méthode de synthèse de haut niveau qui tient compte de la testabilité. La prise en compte de la testabilité en-ligne est effectuée au niveau de la compilation de la description comportementale en graphe de flot de données. Selon les contraintes de test en-ligne, une des méthodes de test en-ligne développées dans le premier axe est intégrée au système au niveau ordonnancement.

Un système numérique défini par sa description comportementale forme l'entrée de la méthode. Dans un premier temps, une optimisation orientée testabilité adresse les équations arithmétiques dans la description comportementale du système. Outre l'amélioration de la testabilité, cette optimisation peut permettre d'améliorer les performances du design final. La description optimisée est compilée en graphe de flot de données ordonnancé. La tâche de la compilation et de l'ordonnancement est résolue par une exploration de l'espace de solutions. Dans cette exploration nous introduisons l'implémentation d'un algorithme génétique adapté à ce type de problèmes. Les contraintes de test en-ligne, de surface et de temps sont considérées dans cette étape. Une fois que le graphe de flot de données ordonnancé est obtenu, la méthode qui répond le mieux aux contraintes de test en-ligne est insérée dans l'ordonnancement nominal du système. L'allocation de ressource et l'assignation permettent la génération d'une architecture testable en-ligne au niveau RTL.

On peut noter que cette nouvelle méthode peut être introduite à plusieurs niveaux dans le flot de la synthèse de haut niveau. Elle peut commencer par une description comportementale avec des contraintes de test en-ligne, de surface et de temps. La description comportementale est optimisée et compilée en graphe de flot de données ordonnancé. Une méthode de test en-ligne est donc insérée au système à ce niveau et l'architecture RTL testable en-ligne est générée. Elle peut également commencer par un graphe de flot de données non-ordonnancé. L'ordonnancement est réalisé en tenant compte des contraintes de test en-ligne, de surface et de temps. Une méthode de test en-ligne est donc insérée au système afin de générer l'architecture testable en-ligne du système.

L'entrée de la méthode peut être finalement un graphe de flot de données ordonnancé. Dans ce cas, le graphe est analysé pour vérifier les contraintes imposées au système et, si ces contraintes sont satisfaites, une méthode de test en-ligne est insérée au système à ce niveau et l'architecture testable en-ligne est générée.

L'étude est illustrée par une implémentation sur les systèmes de traitement numérique de signal. Nous avons traité le cas des filtres numériques récursifs IIR (Infinite Impulse Response). La testabilité en-ligne des différentes structures de réalisation et des différentes

architectures au niveau RTL est analysée. Le chemin de données est généré sous forme d'ASIC (Application Specific Integrated Circuit).

Avant d'entamer l'étude des différentes méthodes proposées de test en-ligne, d'optimisation et de synthèse de haut niveau, une synthèse générale des différents types de systèmes numériques est proposée. L'objectif de cette synthèse est de montrer la position des systèmes de traitement numérique de signal dans les différents type de systèmes numériques et de faire apparaître l'intérêt de l'application de cette étude sur ce type de systèmes. Aussi, une brève introduction à la base théorique du traitement de signal est-elle nécessaire pour la compilation et l'ordonnancement des différentes structures des filtres numériques récursifs. Ce sont le sujet de la suite de cette introduction.

1.1 Classification des systèmes numériques

Les systèmes numériques peuvent être classifiés de plusieurs façons. Selon le point de vue, on peut classifier les systèmes numériques en fonction de leur utilisation, de leur architecture ou bien de la technologie d'implémentation. De manière générale, un système numérique peut être réalisé en deux méthodes principales : de manière complètement câblée comme un circuit intégré spécifique à la demande(*ASICs*) ou basé sur un microprocesseur.

1.1.1 ASIC (Application Specific Integrated Circuits)

Les *ASICs* représentent la réalisation câblée du système où toutes les spécifications fonctionnelles sont implémentées en matériel. Ce type de réalisation est orienté application et n'accepte pas de modifications pour l'adapter à d'autres utilisations.

Les étapes de la synthèse d'un ASIC commencent par la disposition d'une description comportementale ou structurelle du système à réaliser. La description comportementale est compilée en graphe topologique qui représente la dépendance de données. Ce graphe peut être par exemple un graphe de flot de données (DFG) ou un graphe de flot de données et de contrôle (CDFG). Ce graphe est ensuite ordonnancé dans des pas de contrôle tout en respectant un ensemble de contraintes imposées au design final. Les contraintes peuvent être sous forme de nombre de ressources disponibles (contraintes de surface), sous forme de nombre de pas de contrôle ou de durée du cycle fonctionnel (contraintes de temps), sous forme de niveau de consommation du circuit .... Les ressources disponibles sont allouées et une description structurelle au niveau RTL du système est générée.

A partir de la description structurelle, une technologie cible en cellule standard est sélectionnée et les descriptions au niveau porte logique et au niveau layout sont générées.

1.1.2 Microprocesseurs

Les microprocesseurs se caractérisent par la matérialisation sur un circuit intégré (chip) de la fonction d'unité de traitement. Ils se composent, en général, d'une ou plusieurs unités centrales de calcul avec des unités périphériques d'entrée-sortie, de mémoire et de contrôle.

Les microprocesseurs ont connu un succès certain au niveau de la puissance de calcul comme au niveau du marché [44]. En effet, les microprocesseurs représentent le plus fort taux de croissance en matière de puissance de traitement. Cette croissance résulte de la conjonction de plusieurs facteurs, en particulier :

- le niveau d'intégration de la technologie qui permet d'implémenter sur un seul chip un plus grand nombre de fonctions ;

- la capacité de la technologie à fonctionner à des fréquences élevées, l'implémentation sur un seul chip (ou sur un nombre très réduit de chips) permet de tirer le bénéfice maximal puisque les échanges entre chips sont réduits ;

- la diminution des coûts des microprocesseurs permet d'accéder à de nouvelles applications et à de nouveaux marchés et engendre donc des volumes importants. Les revenus ainsi obtenus entraînent des capacités de financement importantes pour le développement des nouvelles générations.

Au niveau du marché, les microprocesseurs représentent un marché actif, environ 86 millions d'unités vers les années 1997/1998. Cette part de marché s'explique par le fait que les microprocesseurs sont utilisés dans des applications à grand volume telles que les fonctions de contrôle dans les produits bruns (téléviseurs, appareils de haute fidélité) ou les produits blancs (électroménager). De plus le phénomène de PC a contribué au succès des microprocesseurs. Aujourd'hui, les microprocesseurs *RISC* ne trouvent d'application que dans des systèmes à hautes performances tels que des stations de travail à orientation scientifique et technique (0.5 millions d'unités par an), des serveurs de base de données ou bien encore des contrôleurs spécialisés (par exemple, accélérateurs numériques et graphiques).

La réalisation d'un système numérique à base de microprocesseurs passe d'abord par une étape de partitionnement qui fixe son architecture matériel/logiciel. Dans cette étapes, les modules constituant l'application sont transposés sur l'architecture choisie. La partie matérielle est représentée essentiellement par les microprocesseurs eux même alors que la partie logicielle est représentée par un programme sauvgardé dans une mémoire appropriée. Il est possible d'utiliser un microprocesseur général, spécialisé ou conçu spécifiquement pour une application déterminée.

Equation de base de la performance

La performance d'un processeur pour l'exécution de la partie traitement s'exprime de la façon suivante :

$$\text{temps par tâche} = \text{instructions par tâche} \times \text{cycles par instruction} \times \text{temps par cycle}$$

Cette équation repose sur l'hypothèse que la tâche n'implique qu'un seul flot d'instructions. C'est généralement le cas mais il existe des approches dans lesquelles on cherche à faire exécuter une tâche donnée sur le plus grand nombre possible de processeurs. Examinons les différents termes de cette équation et les facteurs qui y contribuent :

1. Nombre d'instructions par tâche
   Les facteurs contribuant à ce terme sont :

   (a) l'algorithme que le programme implémente. Un algorithme optimal correspond à la séquence la plus efficace d'instructions exécutées pour la tâche considérée ;

   (b) le niveau d'optimisation du compilateur utilisé pour la traduction du programme. Un compilateur "optimisant" engendre la séquence d'instructions la plus efficace pour un programme donné, ce qui ne signifie pas la séquence la plus courte en terme de nombre d'instructions mais celle qui correspond au temps d'exécution minimal ;

(c) la répartition des fonctions entre les programmes d'application et le système d'exploitation, l'adéquation des primitives du système d'exploitation à supporter les applications ;

(d) l'adéquation de l'architecture au problème à traiter.

2. Nombre de cycles par instruction
   Les facteurs contribuant à ce terme sont :

   (a) le niveau d'optimisation du compilateur : un compilateur optimisant doit sélectionner les séquences d'instructions qui minimisent le nombre moyen de cycles par instruction (séquences qui minimisent les dépendances entre les instructions et qui permettent donc à un processeur pipeline ou super scalaire d'exécuter le flot d'instructions en un minimum de cycles) ;

   (b) la définition de l'architecture : la réduction de nombre des cycles par instruction est l'un des objectifs des architectures, cette direction est illustrée en particulier par les architectures *RISC* ;

   (c) l'implémentation de l'architecture, par exemple pipeline et super pipeline, super scalaire, logique de prédiction des branchements, exécution spéculative...et l'implémentation du système autour de cette architecture, par exemple conception du cache, temps moyen d'accès à la mémoire, débit de la liaison processeur/mémoire...

3. Temps par cycle
   Les facteurs contribuant à ce terme sont :

   (a) la définition de l'architecture : une architecture complexe requiert plus de logique que celle d'une architecture plus simple et peut conduire à une réalisation sur plusieurs chips et donc à des pertes en temps dus aux échanges d'informations entre chips ;

   (b) la technologie : la nature des transistors (CMOS ou bi-CMOS), la taille des circuits (jusqu'à 300 mm^2) ainsi que la finesse du tracé (0.12 μm en 2002) ;

   (c) l'implémentation de l'architecture : la meilleure utilisation du potentiel offert par la technologie compte tenu des objectifs de coût et de performance. Dans ce domaine, on peut citer quelques points sur lesquels l'attention des concepteurs de microprocesseurs doit porter : diminution des effets des temps d'attente dus à la hiérarchie de mémoire, minimisation des effets des branchements, diminution des conflits sur des ressources telles que les registres...

Classification des architectures

Les microprocesseurs peuvent être classifiés en fonction des caractéristiques de l'architecture ou bien en fonction de l'utilisation. Les diffèrents caractères de chaque classe de microprocesseurs seront exposés brièvement dans la suite.

1. En fonction des caractéristiques de l'architecture :

Architecture classique CISC : (Complexe Instruction Set Computer) elle présente un jeu d'instructions riche (à la fois en nombre et en complexité des instructions) et dont la définition n'a, généralement, pas fait l'objet d'une étude systématique mais résulte pour beaucoup d'un processus d'évolution au cours duquel les contraintes de compatibilité ont joué un rôle déterminant.

Architecture à jeu d'instruction réduit RISC : (Reduced Instruction Set Computer) dans cette architecture, le jeu d'instructions a été défini sans contraintes de compatibilité, en général, de façon à minimiser la complexité de l'architecture et dans le but d'obtenir des implémentations de l'architecture les plus efficaces possibles. La minimisation du nombre de cycles par instruction est un objectif des architectures *RISC*. De telles architectures peuvent être vues comme une application du principe de séparation des mécanismes et de la stratégie : ne pas spécialiser le matériel (c'est-à-dire le mécanisme, donc l'architecture du microprocesseur) et laisser au logiciel le soin d'implémenter les fonctions nécessaires (stratégie d'utilisation des mécanismes de base fournie par le microprocesseur).

Architectures spécialisées pour le traitement du signal DSP : (Digital Signal Processor) il s'agit d'architectures définies en fonction des Caractéristiques de ce type d'application. Ces architectures se caractérisent, en général, par les éléments suivants :

- une instruction de type *Multiply and ACcumulate* (*MAC*) permettant, en un seul cycle, de réaliser une multiplication et une addition. On rencontre ce type de calcul dans les produits de matrices et les calculs de séries ;
- des opérations multiples par cycle en particulier pour les boucles : contrôle de la boucle et fonctions d'adressage dans les tableaux ;
- une intégration des fonctions d'accès à la mémoire (*DMA* Direct Memory Access) ;
- des mémoires locales au microprocesseur ;
- un changement de contexte rapide.

Architecture de microprocesseur en réseau : ce type de microprocesseurs est adapté à la réalisation de systèmes permettant d'exploiter certaines formes de parallélisme. Un tel système est composé d'un certain nombre de cellules de traitement, chaque cellule possède ses propres ressources dont une mémoire et communique avec les autres au moyen de messages. Il s'agit de structures *MIMD* (Multiple Instruction stream, Multiple Data stream). Chacun des microprocesseurs est doté, en sus la fonction de traitement, d'une fonction de communication avec l'environnement sous forme de messages. Pour tirer profit de ce type de structure, l'application doit être structurée de façon appropriée : division de l'application en tâches pouvant s'exécuter en parallèle sur différents processeurs tout en minimisant la communication entre ces processeurs.

2. En fonction de l'utilisation :

Usage général : cette dénomination recouvre des applications dans lesquelles le ou les microprocesseurs sont destinés à exécuter des programmes non définis à priori.

Il s'agit de systèmes programmables tels que les stations de travail MS-DOS ou Unix, les serveurs Unix...Pour de telles applications, la compatibilité entre les différentes générations de microprocesseurs ou bien encore entre les fournisseurs de microprocesseurs est une contrainte très forte.

Systèmes préprogrammés (embedded) : cette dénomination recouvre des applications dans lesquelles le ou les microprocesseurs sont destinés à exécuter un ensemble de programmes déterminé au niveau même de la conception. L'utilisateur de tels systèmes n'a pas la possibilité de modifier les programmes, il n'a bien souvent pas conscience d'utiliser un système fondé sur des microprocesseurs. Des exemples typiques se trouvent dans les applications de type contrôleur tels que le système de contrôle de l'allumage et de la carburation d'un moteur d'automobile, le système de contrôle de freinage (antiblocage), l'unité de contrôle d'un avion, le contrôleur d'une imprimante, d'un télécopieur, d'un appareillage électroménager ...Dans ce type d'utilisation, on porte l'accent sur le niveau d'intégration des fonctions au niveau du microprocesseur : la tendance vers le "système sur un chip", le microprocesseur intègre les fonctions essentielles telles que : la mémorisation du programme, la mémoire pour les données, les entrées-sorties spécialisées...Pour de telles applications, la compatibilité n'est pas une caractéristique essentielle et le coût de transposition des programmes n'est pas toujours le facteur déterminant devant les économies d'échelle que peut représenter un niveau d'intégration plus important apporté par une nouvelle génération de microprocesseurs.

Systèmes spécialisés : la destination du système impose des contraintes particulières vis-à-vis du ou des microprocesseurs utilisés. Cette catégorie regroupe à la fois les systèmes à usage général et les systèmes préprogrammés. Par exemple, une station de travail portable impose des contraintes sévères en termes de poids du système et d'autonomie de fonctionnement : cela conduit à des microprocesseurs nécessitant un minimum d'énergie pour assurer leur fonctionnement. Les besoins de l'application peuvent conduire soit à des dérivés "naturels" des implémentations de microprocesseurs existantes (faible consommation, résistance aux rayonnements), soit à des implémentations spécialisées (support des mécanismes de tolérance aux défaillances).

Il y a indépendance entre les deux classifications : caractéristiques de l'architecture et utilisation des microprocesseurs, comme le montrent les exemples suivants :

- architecture *CISC* et usage général : x86 et compatibles utilisés pour la réalisation des PC, x86 utilisés pour la réalisation des serveurs (Sun sur base SPARC, IBM sur base Power, HP sur base PA...) ;
- architecture *CISC* et usage spécialisé : contrôleurs à base 680x0 ou x86. On rencontre dans cette catégorie des processeurs spécialisés dérivés d'architecture *CISC* ;
- architecture *RISC* et usage spécialisé : on rencontre soit des microprocesseurs dérivés d'architecture *RISC* d'usage général mais ayant reçu des extensions spécifiques (gamme d'IDT dérivée de l'architecture R4000 de MIPS), soit des architectures

RISC spécifiquement dédiées à cet usage (gamme 29000 d'AMD, 960 d'Intel, Transputer d'INMOS, l'ARM...).

1.1.3 Microsystèmes

Les microsystèmes sont des composants électromécaniques fabriqués à l'échelle du micron par des procédés technologiques issus de la microélectronique. Ils associent sur un même substrat des capteurs et des actionneurs avec des circuits analogiques et numériques d'interface. Comme pour les circuits intégrés, le test des microsystèmes est une étape importante du cycle de fabrication en terme de coût mais également pour assurer un certain niveau de qualité et de fiabilité.

1.2 Traitement numérique de signal (*DSP*)

Le traitement numérique du signal désigne l'ensemble des opérations, calculs arithmétiques et manipulation de nombres, qui sont effectués sur un signal à traiter, représenté par une suite ou un ensemble de nombres, en vue de fournir une autre suite ou un autre ensemble de nombres, qui représente le signal traité [9]. Les fonctions les plus variées sont réalisables de cette manière, comme l'analyse spectrale, le filtrage linéaire ou non linéaire, le transcodage, la modulation, la détection, l'estimation et l'extraction de paramètres. Les machines utilisées sont des calculateurs numériques.

Les systèmes correspondant à ce traitement obéissent aux lois des systèmes discrets. Les nombres sur lesquels le traitement porte peuvent dans certains cas être issus d'un processus discret. Cependant, ces nombres représentent souvent l'amplitude des échantillons d'un signal continu et dans ce cas, le calculateur prend place derrière un dispositif convertisseur analogique-numérique et éventuellement devant un convertisseur numérique-analogique.

L'essor du traitement numérique date de la découverte d'algorithmes de calcul rapide de la Transformée de Fourier Discrète. En effet, cette transformation est à la base de l'étude des systèmes discrets et elle constitue dans le domaine numérique l'équivalent de la transformation de Fourier dans le domaine analogique, c'est le moyen de passage de l'espace des temps discrets à l'espace des fréquences discrètes. Elle s'introduit naturellement dans une analyse spectrale avec un pas de fréquence diviseur de la fréquence d'échantillonnage des signaux à analyser.

Les algorithmes de calcul rapide apportent des gains tels qu'ils permettent de faire les opérations en temps réel dans de nombreuses applications pourvu que certaines conditions élémentaires soient remplies. Ainsi, la Transformation de Fourier Discrète constitue non seulement un outil de base dans la détermination des caractéristiques du traitement et dans l'étude de ses incidences sur le signal, mais de plus, elle donne lieu à la réalisation d'équipements chaque fois qu'une analyse de spectre intervient, par exemple, dans les systèmes comportant des bancs de filtres ou quand elle conduit à une approche avantageuse pour un circuit de filtrage.

En tant que système, le calculateur de Transformée de Fourier Discrète est un système linéaire discret, invariant dans le temps.

La linéarité et l'invariance temporelle entraînent l'existence d'une relation de convolution qui régit le fonctionnement du système, ou filtre, ayant ces propriétés. Cette relation de convolution est définie à partir de la réponse du système au signal élémentaire que représente

une pulsion, la réponse pulsionnelle, par une intégrale dans le cas des signaux analogiques. Ainsi, si $x(t)$ désigne le signal à filtrer, $h(t)$ la réponse pulsionnelle du filtre, le signal filtré $y(t)$ est donné par l'équation :

$$y(t) = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau$$

Dans les systèmes numériques la convolution se traduit par une sommation. Le filtre est défini par une suite de nombres qui constituent sa réponse pulsionnelle. Ainsi, si la suite à filtrer s'écrit $x(n)$, la suite filtrée $y(n)$ s'exprime par la sommation suivante, où n et m sont des entiers :

$$y(t) = \sum_{m} h(m)x(n-m)$$

Deux cas se présentent alors. Ou bien la sommation porte sur un nombre fini de termes, le filtre est dit à réponse pulsionnelle fini (*FIR*) et se désigne par non récursif car il ne nécessite pas de boucle de réaction de la sortie sur l'entrée dans sa mise en œuvre. Ou bien la sommation porte sur un nombre infini de termes, le filtre est dit à réponse pulsionnelle infinie (*IIR*) ou encore de type récursif, car il faut réaliser sa mémoire par une boucle de réaction de la sortie sur l'entrée. Son fonctionnement est régi par une équation selon laquelle un élément de la suite de sortie $y(n)$ est calculé par la sommation pondérée d'un certain nombre d'éléments de la suite d'entrée $x(n)$ et d'un certain nombre d'éléments de la suite de sortie précédents. Par exemple, si L et K sont des entiers, le fonctionnement du filtre peut être défini par l'équation suivante :

$$y(n) = \sum_{l=0}^{L} a_l x(n-l) - \sum_{k=1}^{K} b_k y(n-k)$$

Les a_l ($l = 0, 1, \ldots, L$) et b_k ($k = 1, 2, \ldots, K$) sont les coefficients. L'étude de ce type de filtre ne se fait pas en général simplement de manière directe. Il est nécessaire de passer par un plan transformé. La transformation en Z est beaucoup mieux adaptée aux systèmes discrets. Un filtre est caractérisé par sa fonction de transfert en Z, désignée généralement par $H(Z)$, et qui fait intervenir les coefficients par l'équation suivante :

$$H(Z) = \frac{\sum_{l=0}^{L} a_l Z^{-1}}{1 + \sum_{k=1}^{K} b_k Z^{-1}}$$

Une autre représentation de la fonction $H(Z)$ est utile pour la conception des filtres et l'étude d'un certain nombre de propriétés, celle qui fait apparaître les racines du numérateur, appelées zéros du filtre, $Z_l (l = 1, 2, \ldots, L)$ et les racines du dénominateur, appelées pôles, $P_k (k = 1, 2, \ldots, K)$:

$$H(Z) = a_0 \frac{\prod_{l=1}^{L}(1 - Z_l Z^{-1})}{\prod_{k=1}^{K}(1 - P_k Z^{-1})}$$

Le terme a_0 est un facteur d'échelle qui définit le gain du filtre.

Il est aussi possible de représenter les filtres numériques par d'autres modèles mathématiques. Ceux-ci correspondent à des équations mathématiques décrivant les relations directes et indirectes entre les entrées et les sorties du système [35], [66] et [28].

Un filtre numérique peut être représenté par les équations suivantes :

$$x(t+1) = A.x(t) + B.u(t)$$

$$y(t) = C.x(t) + D.u(t)$$

Les matrices *A, B, C* et *D* sont de dimensions appropriées. Pour un filtre d'ordre N, la matrice A contient N colonnes et N lignes ($N \times N$) alors que la matrice B contient N lignes et une seule colonne ($N \times 1$). La matrice C contient N colonnes et une seule ligne ($1 \times N$) et la matrice D contient toujours un seul élément (1×1). Le vecteur d'état $x(t+1)$, à un temps quelconque $t+1$, est relié au vecteur d'état précédent $x(t)$ au temps t et à la valeur de l'entrée $u(t)$ à l'instant t. Le vecteur de sortie $y(t)$ à un temps quelconque t est relié au vecteur d'état $x(t)$ et à la valeur de l'entrée $u(t)$ à l'instant t. Ces deux équations représentent les équations d'état d'un filtre numérique.

Ce type d'applications peut être implémenté en logiciel sur des systèmes à base de microprocesseurs généralsou spécialisés ou en matériel par des *ASICs*. Le temps joue un rôle important dans ce type d'applications. La répétition continue d'un ensemble d'opérations implique le traitement en temps réel des informations qui arrivent en permanence. Cela a conduit à des optimisations d'architectures *DSP* pour en augmenter le parallélisme comme par exemple dans les architectures pipelinées et super scalaires [45].

Cette partie de généralité continue dans le chapitre suivant par la présentation de l'état de l'art du test et de la synthèse pour la testabilité. Les techniques de test en-ligne sont aussi résumées dans ce chapitre qui se termine par l'indication de la nécessité et des avantages des différentes méthodes proposées par la présente étude.

La deuxième partie est consacrée à la présentation de deux nouvelles méthodes de test en-ligne. Chapitre 3 propose une méthode de test en-ligne non-concurrent alors que chapitre 4 propose une méthode de test en-ligne semi-concurrent. Les avantages et la faisabilité de ces méthodes sont illustrées sur des exemples.

La troisième partie représente une nouvelle méthode de synthèse de haut niveau pour la testabilité. Deux étapes sont prévues dans cette méthode. La première étape, en chapitre 5, consiste à appliquer, sur la description comportementale du système, une optimisation orientée testabilité. La description comportementale optimisée est compilée, dans une deuxième étape, en graphe de flot de données ordonnancé, chapitre 6. Une des méthodes de test en-ligne, proposées dans la deuxième partie, est intégrée au système à cette étape.

Les méthodes de test en-ligne non-concurrent et semi-concurrent combinées avec la nouvelle méthode de synthèse de haut niveau pour la testabilité représentent une solution intégrée pour la génération de systèmes testable en-ligne. La dernière partie représente l'algorithme général de cette solution intégrée, chapitre 7, avec une implémentation par la conception de plusieurs architectures testables en-ligne d'un filtre IIR elliptique du quatrième ordre, chapitre 8.

Chapitre 2

Conception et test de systèmes numériques

L'objectif de ce chapitre est de résumer les différentes étapes de la synthèse de haut niveau de systèmes numériques et d'exposer les techniques de test et de synthèse pour la testabilité, ce qui nous permettra de centrer notre travail au niveau de tâches à optimiser. D'autre part, les techniques de test en-ligne sont explorées et les principes de nouvelles méthodes de test en-ligne non-concurrent et semi-concurrent sont présentés. Enfin, une nouvelle méthode de synthèse de haut niveau pour le test en-ligne (HLS_OLT) est présentée. Cette méthode de HLS_OLT permet la prise en compte des contraintes de test en-ligne mais aussi celles de surface et de délai. Elle implémente, au niveau ordonnancement, les méthodes de test en-ligne développées.

2.1 Synthèse de systèmes numériques

La synthèse d'un système numérique consiste à réaliser en matériel la fonctionnalité demandée tout en satisfaisant à un ensemble de contraintes qui déterminent les caractéristiques de l'implémentation sous forme de limites, par exemple, sur la surface, la vitesse, la consommation ou la testabilité. Il y a en général deux types de descriptions pour représenter un système numérique. La première forme, dite comportementale, consiste à décrire le comportement du système sans se soucier des détails de sa réalisation structurelle. La deuxième forme, dite structurelle, consiste à décrire la structure du système par des unités fonctionnelles et unités de mémoire interconnectées. Il est possible que ces deux formes de représentation soient utilisées pour décrire un système numérique. Par exemple, la description générale du système est structurelle mais chaque unité fonctionnelle est donnée par sa description comportementale.

Par ailleurs, un système numérique peut être représenté à plusieurs niveaux d'abstraction. La différence entre ces niveaux réside dans la quantité de l'information contenue dans la description. Le premier niveau d'abstraction est le niveau système. A ce niveau, le système numérique concerné se représente sous forme de processus communicant. Le deuxième niveau est le niveau de transfert de registre (RTL) où le système se représente sous forme d'unités fonctionnelles et unités de mémoire interconnectées par des bus et des multiplexeurs. Le niveau suivant est le niveau porte logique où le système est un réseau de portes logiques, de bascules et de registres. Finalement, la représentation au niveau circuit par un réseau de

transistors ou au niveau physique par sa réalisation matérielle au niveau layout. Les étapes de synthèse automatique qui permettent l'obtention d'une implémentation physique du système à partir de sa description comportementale sont principalement deux : la synthèse de haut niveau et la synthèse de bas niveau.

2.1.1 Synthèse de haut niveau (*HLS*) et synthèse *RTL*

La synthèse de haut niveau (*HLS*) [118], accepte comme entrée des spécifications comportementales et comprend des tâches servant à transformer le design du niveau comportemental au niveau transfert de registre (*RTL*). Les tâches du processus de la synthèse de haut niveau peuvent être énumérées de la manière suivante :

Compilation : la première tâche dans un flot de synthèse de haut niveau consiste à compiler une description comportementale donnée en une représentation intermédiaire/interne qui est souvent sous forme de graphe topologique comme le graphe de flot de données (*DFG*) et le graphe de flot de contrôle (*CFG*). Le résultat de cette étape influence fortement les performances du résultat final.

Ordonnancement : la deuxième tâche consiste en l'affectation des opérations du graphe produit par la compilation à des pas de contrôle. Il détermine ainsi les opérations qui doivent s'exécuter en parallèle (durant ce même pas de contrôle). Cette affectation doit respecter plusieurs contraintes, en particulier les dépendances de données et de contrôle, inhérentes à la spécification initiale. A cette étape, le design se distingue en deux parties, la première partie représente le chemin de données (*data path*) sous forme d'un graphe de flot de données (*DFG*), et la deuxième partie représente le contrôle nécessaire pour enchaîner une séquence d'opérations qui réalisent le calcul demandé, sous forme de machine à état fini (*FSM*).

Allocation : cette tâche permet l'assignation des unités fonctionnelles, sélectionnées dans une bibliothèque d'éléments, aux opérations et des registres et éléments mémoire aux variables. Cette allocation se fait généralement en cinq étapes :

- Sélection du type d'unité fonctionnelle.
- Détermination du nombre d'unités fonctionnelles et registres nécessaires.
- Assignation des unités fonctionnelles aux opérations de la description initiale.
- Assignation des registres aux variables de la description initiale.
- Allocation des éléments de routage de données (multiplexeurs et/ou bus).

Extraction du contrôle : la partie contrôle, responsable du chaînage des séquences d'opérations nécessaires à la réalisation du fonctionnement demandé par le design, est générée sous forme de machine à état fini (*FSM*).

2.1.2 Synthèse de bas niveau

La synthèse de bas niveau se décompose en deux étapes, la synthèse logique et le layout. La synthèse logique consiste, à partir d'un ensemble de fonctions booléennes, à obtenir automatiquement une description structurelle d'un bloc combinatoire réalisant ces fonctions

de manière optimisée pour la cible technologique considérée. La synthèse logique a été d'abord définie pour les cellules standards. Plus récemment les techniques de synthèse ont été étendues aux circuits programmables. Actuellement, les outils de synthèse logique sont capables de cibler plusieurs technologies de façon efficace.

Le layout est généralement généré à partir du résultat de la synthèse logique. Il représente la réalisation matérielle du système par des couches de silicium, oxyde et métal de façon à optimiser la surface et les caractéristiques électriques et temporelles du système.

2.2 Test de systèmes numériques

Le test d'un système consiste à vérifier son aptitude à fonctionner suivant les spécifications initiales. Pour ce faire, le système sous test est excité et sa réponse est analysée. Cette excitation peut être sous forme de vecteurs de test spécifiques, générés et appliqués aux entrées du système. Ces vecteurs de test sont définis pour garantir un objectif déterminé du test comme par exemple atteindre une couverture déterminée de fautes en optimisant le temps du test ou en minimisant les dégradations des performances [4].

2.2.1 Génération de test

Les vecteurs de test peuvent être générés principalement de façon aléatoire, pseudo-aléatoire ou déterministe.

Génération aléatoire

Les vecteurs de test sont choisis de façon aléatoire parmi toutes les valeurs possibles d'entrées du système testé. Le nombre de vecteurs de test générés peut être limité par le temps de CPU consommé ou la couverture de faute obtenue. Le problème avec cette technique est essentiellement dans le temps de génération de test et la couverture de faute qui est limitée (normalement entre 50% et 70%) [24] et [100].

Génération déterministe

Le test est généré pour un modèle spécifique de faute. On peut distinguer deux approches :

- Génération orientée à des fautes individuelles.
- Génération orientée à un modèle de faute sans viser des fautes individuelles.

Plusieurs modèles de fautes existent pour la génération du test. Le modèle le plus utilisé serait celui de collage à 0 et à 1 (s_a_0 et s_a_1). La réalisation de ce type de test au niveau circuit peut se faire par une mémoire ROM ou bien par une machine à état fini (FSM).

Génération pseudo-aléatoire

Le test pseudo-aléatoire permet de réduire la complexité du processus de génération de test qui est une procédure de recherche exhaustive [8]. Le fait de laisser tourner un générateur de test jusqu'à ce que toutes les fautes présentes dans le système testé soient couvertes peut devenir une tâche très lourde. Il est souvent utile de coupler la génération de test avec une simulation de faute pour réduire la complexité de la tâche.

Après avoir généré un ou plusieurs vecteurs de test pour une faute spécifique, les vecteurs générés sont appliqués à un modèle du système sous test et les autres fautes sont simulées.

Les fautes qui sont détectées par cette simulation sont considérées testées et sont éliminées de la liste des fautes à détecter. Cela réduit progressivement le nombre des fautes pour lesquelles le générateur de test doit chercher des vecteurs de test. Beaucoup de fautes sont détectées avec les premiers vecteurs de test. Pourtant, quand la couverture de fautes s'approche de 100%, typiquement une seule nouvelle faute est détectée par chaque nouveau vecteur de test.

Plusieurs travaux ont montré que le nombre, relativement élevé, de fautes détectées par chacun des premiers vecteurs de test, est presque indépendant du vecteur généré ou de la faute ciblée. Cela a amené certains à générer initialement un nombre entre 10 et 100 de vecteurs de test de façon pseudo-aléatoire, sans passer par la simulation de fautes, et à simuler ensuite les vecteurs générés. Quand la liste de fautes est convenablement réduite, la procédure conventionnelle de génération de test et simulation de fautes est entamée. On note que le *BIST* (Built-In Self Test) donne une meilleure couverture de faute avec moins de temps de test lorsque le circuit est initialement conçu en tenant compte de la testabilité.

2.2.2 Test structurel et test fonctionnel

Selon la nature des informations disponibles sur le système, un modèle fonctionnel ou structurel peut être utilisé pour la génération du test. Un test structurel nécessite l'accès à la structure du système testé de façon à pouvoir adresser individuellement toutes ses composantes[1]. Pour ce type de test, les notions de contrôlabilité et d'observabilité interviennent au niveau de chaque composant du système pour évaluer sa testabilité. De nombreux algorithmes ont été proposés pour la génération de test structurel [4] et [8]. Cependant, des problèmes restent toujours dans les points suivants :

- les divergences reconvergentes causent des difficultés pour la génération de test (consommation de temps).
- les redondances entraînent l'existence de fautes indétectables qui peuvent :
  - invalider la détéction des fautes qui sont à l'origine détectables ;
  - causer des difficultés pour l'évaluation de la couverture de fautes ; et
  - consommer une grande partie du temps de génération de test en essayant de générer des tests pour ce type de fautes.
- l'estimation du nombre de vecteurs de test nécessaire pour atteindre une couverture déterminée de fautes.

Une méthode différente pour la génération de test consiste à considérer un modèle fonctionnel du système à tester. Dans ce cas, le test, fonctionnel, généré ne nécessite pas l'accès à la structure interne du système, seulement les relations fonctionnelles entre les entrées-sorties du système sont demandées. A noter que certaines techniques de génération de test fonctionnel demandent un accès partiel à la structure du système testé, c'est le cas, par exemple, des techniques de partitionnement (*partitioning techniques*). Le test fonctionnel a les avantages suivants :

- il est indépendant de l'implémentation.

[1] La définition de la composante intervient ici.

- il donne la possibilité de travailler avec des modèles à plus haut niveau d'abstraction comme les arbres de décision binaire (*BDD*) et les modèles graphiques pour les micro-processeurs.

Le test fonctionnel pose un problème au niveau de la détermination de la suffisance du test généré et au niveau de la localisation du composant fautif. Cela revient à la croissance rapide de la complexité des systèmes numériques.

2.2.3 Test en-ligne et test hors-ligne

La méthode classique de test hors-ligne convient pour un test de fin de fabrication ou des tests périodiques pendant la vie de l'équipement [4] et [8]. Pendant ce type de test, le fonctionnement normal du système testé est suspendu pour permettre :

- la reconfiguration du système sous test selon la méthode de test utilisée ;
- l'application de vecteurs de test spécifiques selon l'objectif du test.

Le test en-ligne concerne la vérification du bon fonctionnement d'un système sans affecter son opération normale. Cela peut être pour assurer la préparation du système pour une mission ou un travail critique par exemple. Pour ce faire, trois types de redondances dans le système peuvent être exploités. Ce sont les redondances en information, en temps et en matériel. Ce type de test est appelé concurrent au niveau du système testé.

La redondance en information est une redondance analytique des signaux circulant dans le système. La redondance temporelle est exprimée par les temps libres pendant lesquels une ou plusieures des composantes du système sont oisives et elles peuvent être utilisées pour un autre objectif tel que celui du test en-ligne. La redondance en matériel concerne la disponibilité de plusieurs opérateurs pour assurer l'exécution d'une tâche. Le test en-ligne est donc assuré soit par l'exécution de la même tâche sur tous les opérateurs avec comparaison de résultats soit par le test d'un opérateur pendant que l'autre opérateur exécute la tâche.

Un autre point de vue peut permettre de distinguer trois approches de test en-ligne : test en-ligne concurrent, semi-concurrent, et non-concurrent. Pour le test en-ligne concurrent, les signaux des entrées primaires du système, lors du fonctionnement normal, sont la seule excitation appliquée au système. Ceci élimine la possibilité de pouvoir choisir la séquence des signaux d'entrées pour un objectif déterminé du test. Le test en-ligne concurrent exploite essentiellement la redondance en information contenue dans les signaux circulant dans le système.

Le test en-ligne non-concurrent repose essentiellement sur l'utilisation de la redondance en temps ou en matériel dans le système [52]. Par l'identification et l'utilisation de cette redondance temporelle dans un système, chaque unité fonctionnelle est sélectionnée pendant son temps oisif et un ou plusieurs vecteurs de test lui sont appliqués. La génération, l'application, et la détection en-ligne des erreurs se font par un circuit de test intégré (BIST).

Le test en-ligne semi-concurrent exploite la redondance temporelle dans le système pour vérifier une des propriétés fonctionnelles de l'opérateur sous test. Pour cela, les entrées fonctionnelles qui ont été appliquées à un opérateur pendant son temps fonctionnel sont réutilisées dans le temps oisif de cet opérateur. Ce type de test vise à valider la séquence de calculs qui vient d'être effectuée par l'opérateur.

Ces méthodes seront détaillées ultérieurement dans ce chapitre. Les méthodes de test en-ligne non-concurrent et semi-concurrent sont adressées dans cette étude.

2.3 Amélioration de la testabilité

La testabilité d'un système numérique peut être considérée comme la difficulté [2] de générer et d'appliquer un test sur ce système. Cette difficulté est généralement évaluée par le temps nécessaire pour la génération du test ou pour l'application de ce test sur le système, par le nombre de vecteurs de test générés, ou bien par les dégradations de performances et le coût apportées au système par l'application de la méthode de test. La testabilité peut être évaluée en utilisant les notions de contrôlabilité et d'observabilité. Pour une structure donnée du système, la contrôlabilité évalue la difficulté de générer et d'appliquer une valeur déterminée à un nœud interne du système à partir de ses entrées primaires. L'observabilité évalue la difficulté de cheminer la valeur erronée d'un nœud interne du système jusqu'aux sorties primaires. Cela mène à la définition de la testabilité comme étant la difficulté de générer et d'appliquer un test complet pour un module dans le système. C'est à dire, appliquer les vecteurs de test sur les entrées du module à partir des entrées primaires (contrôlabilité) et observer les résultats aux sorties primaires (observabilité). Au-delà de cette définition classique de la testabilité, il y a quelques propriétés générales qu'un système doit avoir pour être facilement testable [8]. Ce sont : (1) il n'y a pas de redondance logique dans le système, (2) l'horloge est isolée du logique, (3) ses circuits séquentiels sont facilement initialisables, (4) le système ne contient pas de boucles (*self-loops*), (5) les fautes peuvent être facilement localisées, et (6) les dégradations en performances sont minimisées par rapport à un système "normal". Une indication sur l'utilité de chacune de ses propriétés est donnée dans ce qui suit :

1. La redondance logique : une ligne dans un circuit est dite redondante si le fait de coller cette ligne à une valeur (0 ou 1) ne change pas le fonctionnement accompli par ce circuit. Il n'y a pas donc de test qui puisse être généré pour détecter ce type faute pour cette ligne. Cela cause deux problèmes. Le premier est qu'un générateur de test peut tourner pendant des heures en essayant, sans réussir, de trouver un test pour ce type de redondance. Le deuxième est que la présence d'une faute indétectable dans le circuit peut rendre d'autres fautes indétectables alors qu'elles sont détectables sans la présence de cette faute.

2. L'isolation de l'horloge : c'est une règle nécessaire car elle permet au testeur de contrôler l'unité sous test à la vitesse de test plutôt qu'à sa vitesse de fonctionnement normal.

3. Initialisation facile : plus de 80% des vecteurs de test peuvent être nécessaires seulement pour amener un circuit à un état déterminé avant de commencer le test proprement dit. Cette difficulté d'initialisation avant d'appliquer le test peut être évitée si le circuit est équipé d'un moyen simple d'initialisation.

4. Les boucles : au niveau *RTL*, une boucle se forme quand la sortie d'une unité fonctionnelle est connectée directement à une de ses entrées. Les boucles peuvent causer des

[2] ou bien la facilité

problèmes de testabilité au niveau de contrôlabilité et d'observabilité de modules dans une structure. Les vecteurs de test peuvent être partiellement ou même complètement masqués par l'existence de boucles dans la structure.

5. Diagnostic facile : la facilité d'identifier et de localiser les fautes améliore la testabilité et la maintenabilité des systèmes et rend économique l'opération de réparation.

6. Minimisation du coût : les améliorations envisagées de la testabilité d'un système doivent avoir le minimum possible d'impact sur les performances du design final[3].

L'idée de la prise en compte de la testabilité dans les différentes étapes de la synthèse vise à améliorer notre capacité à contrôler ou à observer un nœud interne dans le circuit. Les concepts de contrôlabilité et d'observabilité décrivent efficacement l'objectif de la plupart des approches de conception et de synthèse pour la testabilité. Ces concepts sont applicables aussi bien à la méthode conventionnelle de génération et d'application de test qu'au test intégré (*BIST*).

2.3.1 Conception pour la testabilité (DFT)

Les techniques de *DFT* (*Design For Test*) [4], [8] et [114] ont permis la mise au point de méthodes très efficaces pour résoudre le problème de la testabilité. Ces techniques représentent un ensemble de méthodes de conception utilisées spécialement pour assurer la testabilité d'un système, réduire le coût de génération de test et améliorer la qualité du test généré. Ces techniques visent à améliorer la contrôlabilité et l'observabilité de la structure concernée, par une modification de sa synthèse initiale ou par l'ajout de matériels supplémentaires. Dans ce domaine, plusieurs méthodes sont proposées et utilisées. On peut, cependant distinguer trois catégories principales : les techniques *ad-hoc*, les techniques structurées et les techniques de test intégré (*BIST*) comme l'illustre le tableau 6.2.

Ad Hoc Design Techniques	*Structured Design Techniques*	*Built-In Self Test*
Logic Partitioning	Level-Sensitive Scan Design	Response Analysis
Clock Isolation	Random-Access scan	Exhaustive Testing
Memory Array Isolation	Scan Path	Random Testing
Test Points	Scan/Set Logic	Pre-stored Testing
Bus Access		Functional Testing
Self-oscillation		

Table 2.1: *Les techniques de la conception pour la testabilité (DFT).*

Techniques *ad-hoc*

Les techniques *ad-hoc* sont une collection de méthodes de conceptiopn qui sont appliqées selon le jugement et la compétence du concepteur. Ces techniques ne peuvent pas résoudre complètement le problème de la testabilité. Pour cela les techniques structurées doivent être utilisées.

[3]comparé à un design sans tenir compte de la testabilité.

Techniques structurées

Les techniques structurées se conforment à un ensemble de règles qui sont basées sur la contrôlabilité et l'observabilité des latches dans un système. Si les valeurs de tous les latches sont contrôlables et observables, le problème est réduit au test de blocs combinatoires. Ces techniques sont aussi la base de la réalisation du test intégré (Built-In Test). Beaucoup de structures de test intégré peuvent être réalisées à partir des techniques structurées avec peu de modifications. Cependant, ces techniques sont devenues très coûteuses en temps et en surface parce qu'elles sont, en général, basées sur le principe de *scan register* et ajoutées après avoir réalisé le chemin de données au moins au niveau de transfert de registre (*RTL*). D'une manière générale, les problèmes de ces techniques se résument dans les points suivants :

- l'équilibrage des facteurs affectés (surface, délai, le nombre des entrées-sorties,...) et le gain obtenu ;
- les *scan registers* utilisés sont complexes (surface) ;
- le temps de test par vecteur est augmenté (opérations de shift in/shift out) ;
- une horloge plus lente peut être nécessaire ;
- il y a des systèmes qui ne sont pas facilement réalisables avec cette méthode.

Techniques de test intégré BIST (*Built-In Self Test*)

Comme le nombre de composants qui peuvent être intégrés sur une seule puce de silicium ne cesse daugmenter considérablement, il semble raisonnable qu'une petite partie de ces composants soit dévouée à assurer le test ou le bon fonctionnement du reste du circuit. Ce principe s'appelle "*Built-In Self Test*" (auto-test intégré). Le BIST facilite la génération, l'application, et l'analyse du résultat de test. En plus, il améliore, au niveau RTL, la testabilité des parties difficilement testables du circuit. Le *BIST* peut être structurel ou fonctionnel. Les approches existantes peuvent être implémentées soit avec des vecteurs de test déterministe stockés préalablement soit en exécutant un programme fonctionnel désigné pour tester des parties spécifiques de la structure testée. Un test exhaustif nécessite un partitionnement logique de la structure testée pour rendre le test praticable. Une autre classe de *BIST* structurelle est le test pseudo-aléatoire. Une particularité intéressante de cette classe est le fait que les vecteurs de test pseudo-aléatoire peuvent être générés par des générateurs intégrés simples.

Dans ce contexte, nous prposons dans cette étude des méthodes de test en-ligne basées sur le test intégré (BIST). Ces méthodes utilisent les techniques de test pré-stocké et celles de test fonctionnel.

2.3.2 Synthèse de haut niveau pour la testabilité (HLSFT)

Les performances des équipements industriels croissent avec les progrès technologiques ; simultanément, leur complexité est accrue, rendant ainsi de plus en plus difficiles les tâches de vérification des parties intégrées du système. Ceci impose que les paramètres de testabilité soient pris en compte dès les premières étapes de la conception du système et qu'une

politique de test homogène soit définie aux différentes phases de la vie de l'équipement (conception, fabrication et opération sur site). Le coût de l'amélioration de la testabilité peut être considérablement réduit en tenant compte des contraintes de testabilité dans les différentes étapes de la synthèse de systèmes digitaux. Plus les contraintes sont considérées tôt dans le flot de synthèse, plus le système résultant est meilleur en testabilité et en performance. Les recherches sont actuellement orientées vers l'introduction des contraintes de testabilité lors de la synthèse de haut niveau [34] et [54]. Les styles de synthèse de haut niveau qui donnent un système testable sont regroupés sous le nom de synthèse de haut niveau pour la testabilité *HLSFT* (*High Level Synthesis For Testability*).

Les techniques de *HLSFT* visent l'amélioration de la testabilité par l'augmentation de la contrôlabilité et de l'observabilité des registres, par le soin qu'elles mettent à éviter la formation des boucles (*self-loops*) dans le chemin de données et par la modification de la description initiale du système en ajoutant des instructions ou des opérations de test ou en appliquant quelques méthodes de conception pour la testabilité (*DFT*) au niveau comportemental.

Plusieurs travaux ont été réalisés dans ce domaine. La plupart de ces travaux exploitent les principes de *F-path* (Fault-path), *S-path* (Stimulus-path), *T-path* (Transformation-path) ou *I-path* (Identity-path) [33]. Ces principes sont inspirés de l'idée d'utiliser les blocs fonctionnels pour assurer le test (chemin d'application de vecteurs de test et chemin d'observation de résultats) des différentes parties du système sans avoir à introduire de matière supplémentaire pour les techniques de *DFT*. Une unité fonctionnelle peut être rendue transparente pour les données de test assurant ainsi d'une part la propagation des vecteurs de test vers les unités fonctionnelles en aval (contrôlabilité) et d'une autre part la propagation des résultats de test des unités en amont (observabilité). Deux propriétés des unités fonctionnelles doivent être garanties pour faciliter le test complet. Ce sont : "*one-to-one mapping*" et "*onto mapping*". La propriété "*one-to-one mapping*" (notion de *F-path* (Fault-path)) garantie que toutes les données présentes à la sortie de l'unité fonctionnelle en amont soient exactement transmises aux entrées de l'unité fonctionnelle en aval. Ceci assure une observabilité parfaite pour les unités fonctionnelles en amont. La propriété "*onto mapping*" (notion de *S-path* (Stimulus-path)) garantie que toute donnée peut être générée aux sorties de l'unité fonctionnelle concernée. Cela assure une contrôlabilité parfaite pour les unités fonctionnelles en aval.[4]

Les différentes approches considèrent la testabilité à partir d'une représentation graphique du système sous forme de graphe de dépendance de données (Data Flow Graph (*DFG*)) ou graphe de dépendance de contrôle (Control Flow Graph (*CFG*)). La plupart des approches supposent avoir un graphe ordonné et introduisent la testabilité au niveau allocation de ressources et assignation sous forme d'amélioration de la contrôlabilité et de l'observabilité des registres internes [32], [68] et [69].

Par exemple, l'approche proposée par Flottes *et al.* [32] est basée sur la transparence du chemin de données qui permet de trouver un chemin permettant l'application des vecteurs de test (contrôlabilité) à partir des entrées primaires du système et un chemin permettant l'observation des résultats du test (observabilité) aux sorties primaires. Cela est fait en garantissant la propagation de données via les différentes unités fonctionnelles pour assurer l'acheminement des vecteurs de test aux parties internes dans la structure à partir des entrées

[4] L'exploitation des principes de *F-path* et *S-path* élimine le besoin de fixer "other inputs"...

primaires du système et assurer la propagation du résultat de test des différentes unités fonctionnelles aux sorties primaires du système. La testabilité est considérée à la dernière tâche de la synthèse de haut niveau, à savoir à l'assignation des variables aux registres et à l'interconnexion.

Majumdar *et al.* [68] et [69] proposent une approche visant à éviter la formation de boucles et à améliorer la contrôlabilité et l'observabilité des registres qui sont difficilement contrôlables et observables. Cela se fait à l'étape de l'assignation de variables aux registres. En plus de considérer la testabilité à la dernière tâche de la synthèse de haut niveau, comme [32], le problème de l'assignation des variables aux registres est formulé comme un "network flow model" où les nœuds représentent les modules[5] et les opérations[6] tandis que les arcs représentent les possibilités de l'assignation d'une opération à un module/registre. Le réseau des opérations et modules/registres ainsi construit est résolu par la méthode "simplex". Pour minimiser la complexité de la solution, ce réseau est construit et résolu séparément pour chaque pas de contrôle. Néanmoins, ce type de solutions ne peut pas garantir la solution optimale.

De même, dans [32], l'auteur propose une analyse de testabilité basée sur la notion de transparence et considérée au niveau d'assignation de variables aux registres. L'idée est que l'assignation simultanée de registres à des variables qui sont contrôlables/observables et à d'autres variables qui ne sont pas contrôlables/observables améliore la testabilité de ces dernières.

Dans l'approche proposée par Steensma *et al.* [107], la testabilité est considérée à l'étape de la conception et du choix de l'unité fonctionnelle qui garantit la transparence du chemin de données au niveau *RTL*. Le problème de l'assignation d'unités fonctionnelles dédiées à l'application est représenté par un graphe de compatibilité où les nœuds représentent les groupes d'opérations qui peuvent être exécutées sur la même unité fonctionnelle et les arcs sont pondérés par les mesures de compatibilité entre les groupes. Ces mesures représentent la différence en surface entre une unité fonctionnelle capable d'exécuter deux groupes i et j et deux unités fonctionnelles dédiées à chaque groupe. Ce graphe est utilisé pour réunir les différents groupes en utilisant les mesures de compatibilité qui comprennent des analyses de testabilité.

Il y a, toutefois, des travaux qui considèrent la testabilité lors de l'ordonnancement. Par exemple, dans le travail de Vahidi *et al.* [110], des matrices de testabilité sont utilisées pour réaliser une sorte de restructuration d'un ordonnancement donné. Cette restructuration consiste simplement à changer les positions des points de mémorisation qui déterminent les différents pas de contrôle. Il est clair qu'une partie importante d'autres ordonnancements possibles est ignorée et la possibilité de tomber sur un optimal local est très forte et dépend fortement de la technique utilisée pour réaliser l'ordonnancement. Un autre travail [49] s'intéresse à la testabilité du chemin de données durant l'ordonnancement et l'assignation. La mesure de testabilité proposée est applicable au niveau structure *RTL*. Il y a donc besoin d'avoir une idée de la structure afin de pouvoir évaluer la solution. Pour ce faire, un pré-ordonnancement est réalisé en utilisant la méthode de "*Force-directed scheduling*" [87] pour estimer le nombre des unités fonctionnelles nécessaires à la réalisation du chemin de

[5] alloués
[6] ordonnées

données. Basées sur cette estimation, des contraintes de partage des unités fonctionnelles entre les opérations sont imposées de façon à améliorer la testabilité. L'ordonnancement est donc refait par la même méthode "*Force-directed scheduling*" en tenant compte de ces nouvelles contraintes. Cette approche est coûteuse au niveau temps de calcul en raison de l'utilisation répétée de l'ordonnancement "*Force-directed scheduling*". Les solutions proposées pour l'ordonnancement sont, par ailleurs, limitées entre les ordonnancements le plutôt possible (*ASAP*) et le plus tard possible (*ALAP*) qui représentent respectivement les bornes supérieures et inférieures des solutions possibles avec la méthode d'ordonnancement "*Force-directed scheduling*". Il faut noter qu'aucune des méthodes précédemment exposées ne considère les contraintes de testabilité avant l'étape de l'ordonnancement. En effet, elles partent d'un graphe de flot de données ordonnancé et visent à modifier cet ordonnancement par l'exploitation d'une des méthodes qui améliorent la testabilité.

2.3.3 Mesures de testabilité

Il est possible de générer une estimation de la testabilité pour un circuit sans réaliser la génération de vecteurs de test et la simulation de fautes. Pour le test déterministe, il existe des mesures de testabilité qui donnent une comparaison relative entre les coûts de la génération de test et la simulation de fautes. Ces mesures aident aussi à déterminer les zones du circuit où la génération de test sera difficile et où les techniques de *DFT* doivent être utilisées. Pour le test aléatoire, il existe des mesures qui donnent la probabilité de détecter une faute dans le circuit de façon à pouvoir identifier les zones qui sont difficiles à tester pour les modifier et de façon à estimer la qualité d'un test donné [100] et [24]. Pour cela, une mesure de testabilité d'un circuit est utile seulement si le coût de la génération de cette mesure est nettement inférieur au coût de la génération de vecteurs de test et la simulation de fautes. Cela implique que les mesures de testabilité ne donnent qu'un résultat approximatif. Pourtant, les mesures de testabilité peuvent être utilisées pour guider le processus de synthèse dans ses différentes étapes. Par ailleurs, elles peuvent aider dans l'exploration de l'espace de solutions en vue de l'identification des optimales potentielles.

Les approches traditionnelles ont beaucoup utilisé les mesures de testabilité pour aider dans le processus de la génération de test. Au niveau porte logique, les notions de 0/1 contrôlabilité et observabilité ont été extensivement utilisées. Au niveau fonctionnel[7], les mesures traditionnelles de testabilité ne sont plus significatives. Pour cela, des mesures fonctionnelles de testabilité ont été développées [20] et [109]. A la place de savoir à quel degré un octet dans un mot est-il contrôlable/observable, le besoin est de savoir à quel degré est-il facile de produire une valeur déterminée à l'entrée d'une unité fonctionnelle[8]. En plus, les mesures de testabilité au niveau fonctionnel se trouvaient adaptées pour répondre aux besoins des styles d'architectures non-traditionnelles comme l'architecture "*bit-serial*". Ce type de mesures de testabilité a été développé pour refléter la testabilité d'un circuit numérique au niveau transfert de registre (*RTL*) même si elles étaient appliquées sur des représentations graphiques des systèmes considérés.

[7]Soit niveau *RTL*

[8]Pour la justification et la propagation.

2.4 Techniques de test en-ligne

Les techniques de test en-ligne sont inévitables pour les applications qui nécessitent de montrer l'état du système lors du fonctionnement normal. Les futurs systèmes doivent inclure des moyens d'autotest qui implémentent les techniques de test en-ligne telles que les techniques de redondance du matériel, de redondance du temps, et de redondance de l'information (techniques de codage).

Les techniques de redondance du matériel comptent parmi les techniques primitives utilisées pour la détection en ligne des fautes des systèmes numériques. Pour ces techniques, des copies du système à tester sont utilisées [82], [93], et [99] et les sorties des copies sont connectées à un élément de vote de majorité. Avec n copies identiques du système, le schéma de la redondance du matériel peut tolérer $n/2$ copies fautives. Dans le cas de la redondance dynamique, les copies fautives, lorsqu'elles tombent en panne, sont remplacées automatiquement par de nouvelles copies.

Les techniques de redondance du temps sont basées sur la répétition des opérations nominales dans les périodes de repos des unités fonctionnelles du circuit sous test. Les avantages de ces techniques, par rapport aux autres techniques, sont : le matériel supplémentaire nécessaire pour l'implémentation est négligeable et la dégradation de performances du circuit sera banalisée. Pour ces techniques, chaque opération est répétée plusieurs fois avant que le circuit ne délivre le résultat final. Les sorties à chaque fois sont stockées pour la comparaison [21], [61], et [93]. Par exemple, un additionneur peut être testé en répétant l'opération de l'addition trois fois [61]. D'abord, l'opération d'addition est effectuée sur les données A, B selon la relation de sortie $S = A + B$ et la valeur de sortie obtenue est stockée. La deuxième opération sera effectuée selon la relation $S = B + A$ et la nouvelle valeur de sortie est comparée avec la valeur stockée. Chaque différence est marquée et stockée. La troisième opération d'addition sera effectuée selon la première relation $S = A + B$ et l'opération de comparaison se répète en remarquant s'il y a un désaccord entre la valeur calculée et la valeur stockée. S'il n'y a pas de désaccord, alors le résultat obtenu à la première étape peut être utilisé, il n'y a pas de faute. S'il y a des désaccords, alors il est possible de déterminer les bits erronés et de les corriger.

Les techniques de redondance de l'information, ou techniques de codage, effectuent la même idée de base : celle de transformer le mot de données par des opérations logiques sur les bits de données en un autre mot codé et augmenté en taille. Parmi les codes les plus connus, on compte les codes de parité [38], [90], et [93] et les codes de résidu [2], [101], [76], et [93]. Chaque mot codé comporte les bits originaux de mot de données (bits d'information) et les bits de vérification (check bits) résultant de l'opération de codage. Les bits de vérification sont construits pour examiner les bits d'information.

Les techniques de redondance du matériel sont très coûteuses en surface. Les techniques de redondance de l'information restent parmi les moyens les plus importants utilisés aujourd'hui pour le test en-ligne des systèmes numériques. Le problème avec ces techniques réside dans la dépendance, de la qualité du test, des informations reçues sur les entrées du système. Cela rend ces techniques pauvres en couverture de faute. Les techniques de redondance du temps peuvent pallier ce problème par deux aspects : l'application du test déterministe dans les temps oisifs des opérateurs et la vérification du calcul qui vient d'être effectué sur le système. La présente étude est basée sur l'exploitation de la redondance du

temps pour réaliser le test en-ligne. Cela nous conduit à étudier, plus en détail, des techniques de redondance du temps. Plusieurs méthodes sont exposées, comparées, et adoptées par cette étude. Ces méthodes concernent deux catégories de test en-ligne : le test en-ligne non-concurrent et le test en-ligne semi-concurrent. Elles conviennent aux environnements où l'occurrence de faute permet la vérification périodique du résultat toutes les Nème itérations du processus à la place de la vérification à la fin de chaque itération.

2.4.1 Test en-ligne concurrent

La méthode concernée de test en-ligne concurrent [2] et [101] exploite essentiellement la redondance analytique décrivant les relations entre les histoires des signaux d'entrées et de sorties du système sous test. Cette méthode est applicable aux systèmes numériques linéaires.

L'objet de ce travail est d'étudier une nouvelle approche de conception et d'intégration des détecteurs de défauts en-ligne. La méthode garanti aussi une détection robuste de fautes, c'est à dire une sensibilité maximale pour les fautes et minimale pour le bruit généré à l'intérieur des systèmes.

Le fait que cette méthode adresse seulement les systèmes numériques linéaires limite un peu son champ d'application. L'exploitation de la redondance analytique dans le système fait intervenir la complexité du système sur la méthode de test en-ligne. Les méthodes de test en-ligne non-concurrent et semi-concurrent permettent de contourner cette difficulté par l'exploitation de la redondance temporelle dans le système sous test.

2.4.2 Test en-ligne non-concurrent

Le test en-ligne non-concurrent, au niveau unité fonctionnelle, vise à appliquer des vecteurs de test sur les unités fonctionnelles pendant leurs temps oisifs. Le système doit être équipé d'une unité de génération, d'application et d'analyse de résultat de test.

Dans [103] (Singh et Knight) on propose une méthode de test pseudo-aléatoire intégré. A partir d'un graphe de flot de données ordonnancé, un schéma de test en-ligne est inséré pour assurer l'application des vecteurs de test sur les opérateurs. Le schéma de test doit comprendre tous les chemins de l'ordonnancement nominal du système.

Le test en-ligne non-concurrent convient plutôt aux fautes permanentes. Il n'a pas été suffisamment adressé pour le test déterministe. L'un des objectifs de nos travaux consiste à évaluer ce type de test en-ligne pour le test déterministe.

Dans ce contexte, nous proposons une méthode de test non-concurrent intégré. Une unité de génération, d'application et d'analyse de résultat de test est présentée. Les coûts en surface et en délai de l'implémentation de la méthode sont évalués. Un algorithme d'insertion des opérations de test dans les temps oisifs est présenté.

Les avantages de notre méthode par rapport à celle présentée par Singh et Knight [103] résident dans deux points. Le premier est que nous proposons un test déterministe intégré, ce qui signifie qu'une couverture élevée de faute est garantie. Le deuxième et que la testabilité en-ligne est prise en compte au niveau de la compilation, i.e., au niveau du choix du graphe de flot de données. Ce qui améliore la qualité du test.

2.4.3 Test en-ligne semi-concurrent

Les méthodes de test en-ligne semi-concurrent utilisent les entrées/sorties fonctionnelles du système pour appliquer des excitations sur les différents opérateurs. Ces excitations sont

appliquées pendant les temps oisifs des opérateurs. Un des points communs entre ce type de méthodes et les méthodes de test en-ligne concurrent est l'utilisation des entrées/sorties fonctionnelles du système (ou de l'opérateur). Une différence réside dans le fait que les méthodes de test en-ligne semi-concurrent ne considèrent pas analytiquement l'information continue dans ces entrées/sorties. En effet, les bits récoltés des entrées/sorties des opérateurs sont considérés comme une longue séquence de vecteurs de test aléatoire. Cependant, ce type de test en-ligne peut viser la validation d'une des propriétés fonctionnelles de l'opérateur testé. Le test semi-concurrent est considéré efficace pour les systèmes qui fonctionnent avec un flot continu et varié de données. Ce flot de données peut être considéré comme une séquence très longue de vecteur de test aléatoire qui mène à une couverture de faute acceptable.

Dans cette section, deux types de méthodes de test en-ligne semi-concurrent sont exposées. La première consiste à appliquer un test fonctionnel par l'exploitation de la distributivité de la multiplication par rapport à l'addition. La deuxième méthode [6] consiste à refaire le calcul qui a été effectué sur le système mais dans les temps oisifs des opérateurs et en permutant les unités fonctionnelles. Les résultats des deux calculs sont comparés pour la détection des erreurs.

Pour appliquer un test fonctionnel basé sur l'exploitation de la distributivité de la multiplication par rapport à l'addition, il faut disposer des entrées et des sorties fonctionnelles de l'opérateur considéré. Ces entrées et sorties sont décalées à droite ou à gauche. Le nombre de bits et la direction du décalage peut dépendre de la nature et de la position de l'octet visé par le test. En fait, vu la considération des entrées fonctionnelles comme une longue séquence de vecteurs de test aléatoire, on peut fixer le décalage à un octet à gauche et la diversité des entrées fonctionnelles garantira une bonne couverture de faute. Après le décalage, l'opération est réalisée sur le même opérateur par les entrées décalées. Le résultat est comparé à la sortie fonctionnelle décalée. En cas de différence entre les deux réponses, une erreur est signalée.

La méthode de la vérification globale du calcul consiste à refaire le calcul qui vient d'être effectué sur le système nominal et à comparer le résultat avec le résultat nominal. Pour cela, les unités fonctionnelles sont réutilisées, quand c'est possible, dans leurs temps oisifs. Une allocation d'unités fonctionnelles, différente de celle nominale, est effectuée pour augmenter l'efficacité du test (la probabilité de faire manifester la faute). Le test effectué par cette méthode est un test fonctionnel qui est indépendant de la technologie.

Cette méthode adresse les systèmes à un faible taux d'occurrence de fautes. Cela rend raisonnable le fait que le test ne soit pas réalisé de façon concurrente avec chaque itération du système mais périodiquement toute la n^{eme} itération (n dépend de l'application et du taux de l'occurrence de fautes).

A partir de ces deux méthodes, nous proposons une nouvelle méthode de test en-ligne semi-concurrent. Elle consiste à combiner les principes des deux méthodes pour améliorer la qualité du test. Les avantages apportés par notre méthode résident, essentiellement, dans trois points :

- La généralisation de la première méthode pour considérer globalement le système.
- La possibilité de tester des architectures contenant une seule unité fonctionnelle pour chaque type d'opérations où la permutation des opérateurs n'est pas possible ;

- L'exploitation des variante-duales du graphe de flot de données pour le test en-ligne.

Le tableau (2.2) résume la différence entre les méthodes de test en-ligne qui ont été exposées dans cette section.

Test en-ligne	*Excitation*	*Application*
Concurrent	Entrées/sorties fonctionnelles	Détecteur intégré
Semi-concurrent	Entrées/sorties fonctionnelles	Temps oisifs
Non-concurrent	Vecteurs de test (BIST)	Temps oisifs

Table 2.2: *Méthodes de test en-ligne.*

La figure (2.1) représente une synthèse des techniques de test en-ligne et montre les méthodes adressées par la présente étude.

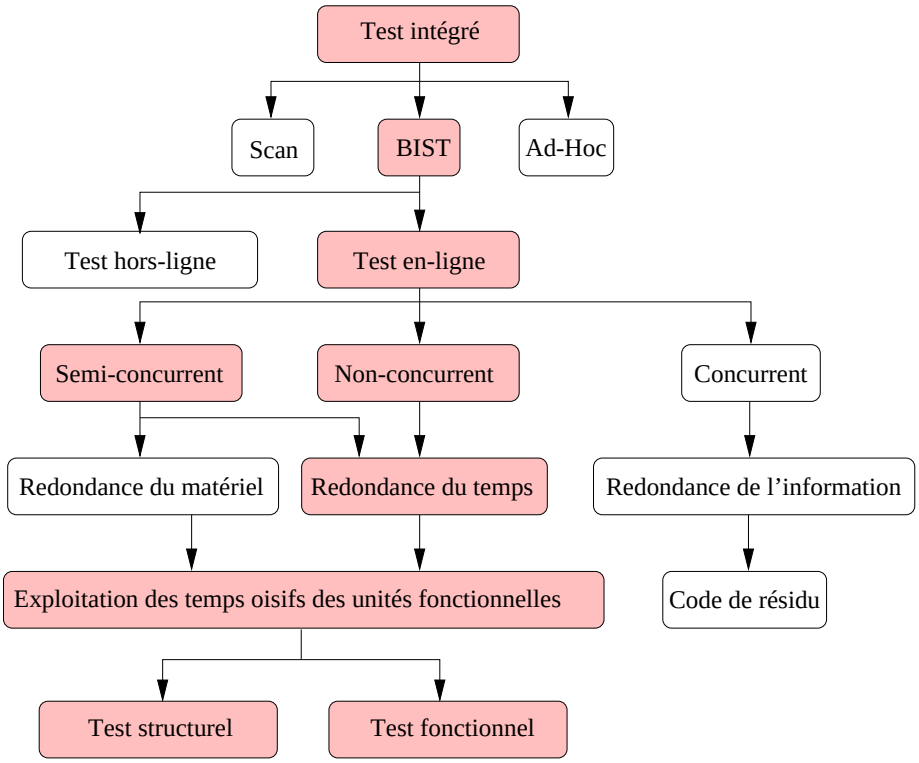


Figure 2.1: *Les techniques de test intégré / Test en-ligne.*

2.5 Nouvelle solution pour HLS_OLT

La présente étude s'intéresse au développement d'une nouvelle méthode de synthèse de haut niveau pour la testabilité en-ligne (High Level Synthesis For On-Line Testability). Les

méthodes de test en ligne non-concurrent et semi-concurrent sont adressées. La méthode proposée favorise l'existence des temps oisifs pendant l'intervalle fonctionnel du système et les exploite pour appliquer des excitations sur les unités fonctionnelles. Avant l'étape de la synthèse de haut niveau, une optimisation de la description comportementale permet d'améliorer la testabilité du système et d'enrichir l'espace de solutions. Les contraintes de testabilité sont considérées dans les deux premières tâches dans le flot de la synthèse de haut niveau, notamment lors de la compilation de la description comportementale en graphe de flot de données (*DFG*) et l'ordonnancement.

La difficulté de manipuler un tel problème vient particulièrement des points suivants :

- Plusieurs tâches dans le flot de la synthèse de haut niveau sont reconnues comme des problèmes NP-complexe. Ainsi, plusieurs approches divisent le problème principal en plusieurs sous-problèmes et résolvent séparément chacun des sous-problèmes pour donner une solution globale au problème principal. Il est connu qu'une telle solution ne garantit pas la solution optimale.[9]

- Il est nécessaire de développer une heuristique pour résoudre le problème de la compilation et l'ordonnancement. Une telle heuristique doit être capable de manipuler, en temps raisonnable, les problèmes d'optimisations très complexes dans le domaine de CAO/VLSI. Cela nécessite le développement de méthodes qui permettent une exécution en parallèle de l'algorithme sur un réseau local.

- Avant l'ordonnancement, pratiquement aucune information n'est pas disponible sur la structure du système en cours de conception. Pour cette raison, les notions traditionnelles de contrôlabilité et d'observabilité ne sont plus valides avant cette étape. Il convient de développer de nouvelles notions d'observabilité et de contrôlabilité qui permettent d'évaluer la testabilité de la future structure RTL à la fin de l'ordonnancement au plus tard.

Avant d'entamer la procédure de la synthèse de haut niveau, une optimisation de la description comportementale permet d'apporter un gain important en testabilité du circuit final. La figure (2.2) représente les étapes principales de la méthode proposée. L'entrée de l'outil développé est une description comportementale du système à synthétiser. La sortie de cet outil étant la description en VHDL de la structure RTL qui répond aux contraintes de testabilité, délai et surface.

2.5.1 Optimisation des temps oisifs

Une description comportementale qui fournit l'entrée d'un outil de synthèse contient un ensemble de constructions de langage de haut niveau (i.e. branche conditionnelle, boucle, ...). Ce type de description utilise des types complexes de données (i.e. entier, matrice, registre, ...). La plupart de systèmes de synthèse associent à chaque construction du langage de haut niveau une structure matérielle déterminée. En raison de la relation très étroite entre les constructions de la description comportementale et les structures matérielles correspondantes, la qualité du design résultant de la synthèse dépend fortement du style de

[9] On peut tomber sur une solution qui ne satisfait pas les contraintes imposées et dire qu'il n'existe pas de solutions pour le système considéré alors qu'une autre solution qui satisfait aux contraintes existe.

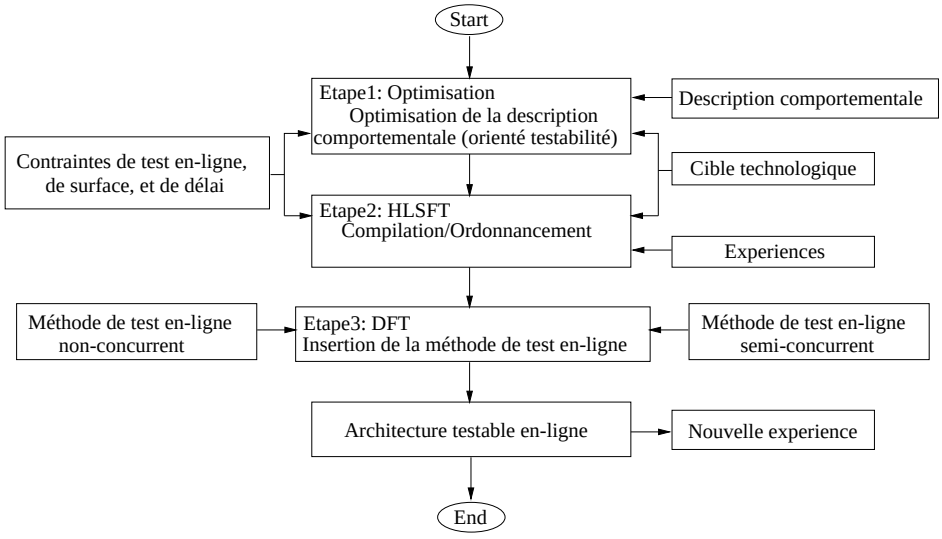


Figure 2.2: *Le flot général de la nouvelle méthode de HLS_OLT.*

la description comportementale. Autrement dit, deux descriptions comportementales qui sont sémantiquement équivalentes mais qui se diffèrent en syntaxe, peuvent donner lieu à deux designs qui sont largement différents. Cela nécessite une optimisation de la description comportementale qui minimise l'effet de la syntaxe sur le résultat de la synthèse. Plusieurs approches ont été proposées pour inclure ce type d'optimisation dans le flot de la synthèse de haut niveau [17], [85], [37], [19], [70], [42], [7], et [72].

A cette étape, une optimisation qui s'applique aux blocs fonctionnels contenant des opérations arithmétiques est proposée par cette étude. Basée sur la factorisation des équations arithmétiques, cette optimisation est orientée testabilité. Elle favorise l'existence de temps oisifs au niveau ordonnancement et permet l'élimination de boucles au niveau transfert de registre (RTL).

La description comportementale est décomposée en plusieurs modules concurrents. Chaque module est représenté par un graphe biparti appelé Graphe de Dépendance de Donnée (DDG). Ce graphe est utilisé pour générer la Matrice de Corrélation de Variables (VCM[ ]) qui est à son tour utilisée pour identifier les meilleures variables communes pour réaliser la factorisation. Cette matrice est utilisée également pour analyser l'existence de boucles potentielles dans les équations factorisées. Ces analyses utilisent une base de données contenant des informations de la cible technologique visée.

2.5.2 Compilation et ordonnancement

A ce stage de la synthèse, une représentation graphique du système est générée par compilation de la description comportementale en graphe de flot de données (DFG) et l'ordonnancement est réalisé. Le problème de la compilation et de l'ordonnancement est formulé comme un problème d'exploration d'espace de solutions. Ce problème est ensuite résolu par l'application d'une heuristique adaptée, basée sur l'idée des algorithmes génétiques (GAs). La prise en

compte des contraintes de testabilité pour le test en-ligne est effectuée lors de la compilation et l'ordonnancement.

Algorithmes génétiques (GAs)

Le domaine des algorithmes génétiques a été développé au début des années 70, mais seulement depuis 1990 que des applications réelles qui démontrent un potentiel commercial ont eu lieu. Ce type d'algorithme a été développé pour simuler quelques processus observés dans l'évolution naturelle. Cette évolution prend place dans les chromosomes ; les engins organiques pour décoder la structure d'un être vivant. Un être vivant est créé partiellement à travers un processus de décodage des chromosomes. Quelques caractéristiques du processus de codage et décodage des chromosomes et qui sont largement acceptées sont [25] :

- l'évolution naturelle est un processus qui agit plutôt sur les chromosomes d'un être vivant que sur l'être vivant lui-même.
- le processus de la sélection naturelle est le lien entre les chromosomes et la structure qu'ils encodent. Ce processus cause la reproduction plus souvent des chromosomes qui encodent de "bonnes" structures que ceux qui ne le font pas.
- le processus de la reproduction est le point auquel l'évolution prend place. La mutation peut causer que les chromosomes des enfants soient différents de ceux de leurs parents biologiques. La recombinaison peut causer la production de chromosomes assez diffèrent en composant des matériels des chromosomes de deux parents.

Ces caractéristiques de l'évolution naturelle ont intrigué John Holland au début des années 70. Il a donc commencé à travailler sur des algorithmes qui manipulent des chaînes de chiffres binaires, des 1 et des 0, qu'il appelait chromosomes. Ces algorithmes ont résolu le problème de trouver les bons chromosomes par la manipulation des chaînes qui les représentent. Aucune information n'était pas nécessaire autour du problème traité. La seule information qui était disponible était l'évaluation de chaque chromosome produit. Et la seule utilisation de cette information était pour orienter la sélection de façon à favoriser la production des bons chromosomes.

Ces algorithmes, utilisant simplement des mécanismes d'encodage et de reproduction, ont montré un comportement compliqué. Ils ont été exploités pour résoudre des problèmes qui sont très compliqués sans avoir une connaissance du monde décodé. C'était une simple manipulation de simples chromosomes. Récemment, l'utilisation des descendants de ces algorithmes a montré qu'ils peuvent évoluer de meilleurs designs, trouver de meilleurs ordonnancements et produire meilleures solutions pour d'autre type de problèmes importants qui ne pourraient pas être résolus aussi bien par d'autres techniques [25] et [71].

Les chromosomes contiennent toutes les informations nécessaires pour déterminer les caractéristiques de l'individu. La reproduction implique, en général, deux parents. Les chromosomes des individus de la nouvelle génération sont composés des portions des chromosomes des parents. Ainsi, les nouveaux individus héritent des caractéristiques de leurs parents. Les algorithmes génétiques utilisent ce mécanisme d'héritage pour résoudre de nombreux problèmes d'optimisation.

L'objectif d'un algorithme génétique est de trouver la solution optimale. Comme ce type d'algorithmes n'est qu'une heuristique, la solution optimale ne peut pas être garantie.

Néanmoins, les expériences ont montré que ces algorithmes sont capables de trouver de bonnes solutions pour des problèmes de types très variés.

Les algorithmes génétiques fonctionnent par l'évaluation d'une population d'individus sur plusieurs générations. Une valeur de *fitness* est associée à chaque individu. Les individus d'une génération sont croisés pour générer de nouveaux individus. Ces nouveaux individus sont mutés avec une basse probabilité de mutation. La nouvelle génération peut être formée complètement des nouveaux individus ou d'une combinaison des nouveaux et des anciens individus. Les nouveaux individus peuvent remplacer complètement les anciens individus.

Les raisons pour lesquelles ce type d'algorithmes est adopté peuvent être résumées par les points suivants :

- Les tâches de la synthèse de haut niveau sont connues comme NP-difficiles. Il est donc nécessaire d'utiliser, pour les systèmes complexes, une méthode d'exploration d'espace de solutions qui est à la fois efficace, robuste et pratique. Les algorithmes génétiques ont montré une efficacité dans la résolution de problèmes très complexes.

- La disponibilité des réseaux locaux est, aujourd'hui, un avantage qui doit être exploité par les outils de synthèse pour améliorer ses performances. Il faut donc une méthode qui permet la répartition de la tâche globale sur plusieurs sous-tâches indépendantes pour résoudre ces sous-tâches individuellement sur une partie du réseau. Les algorithmes génétiques possèdent un parallélisme intrinsèque qui leur permet de tourner sur un réseau local faiblement connecté.

- Pour des systèmes complexes, il convient de garder une trace de la solution trouvée sous forme d'expérience pour guider les explorations futures. Cette base de données d'expériences doit être intelligente (meilleure population initiale) et raisonnable en taille (sous forme génétique). Les algorithmes génétiques sont adaptatifs et apprennent des expériences.

Technologie cible

Les calculs effectués dans les différentes étapes de cette méthode nécessitent la disposition des informations concernant la cible technologique visée. Pour cela, une base de données est établie pour la bibliothèque de cellules standards choisie pour l'implémentation. Les éléments de cette base de données sont la surface et le délai des différents types d'opérateurs. Elle contient aussi, pour les unités fonctionnelles, un ensemble de vecteurs de test qui garantissent un test complet généré automatiquement par l'outil design_analyzer de SYNOPSYS. Ces éléments sont utilisés pour :

- estimer le nombre de boucles potentielles dans une fonction arithmétique.

- calculer le nombre d'opérateurs nécessaires au niveau RTL et définir le pas de contrôle.

- établir les éléments nécessaires pour le calcul de la testabilité du système.

Quelques éléments de cette base de données sont représentés graphiquement dans la figure (2.3). On trouve dans cette figure l'estimation de surface et de délai pour un multiplieur,

un additionneur, un registre et un multiplexeur pour les différents largeurs en bit de ces éléments.

La partie suivante va présenter de nouvelles méthodes de test en-ligne. Ce sont une méthode de test en-ligne non-concurrent, chapitre 3, et une méthode de test en-ligne semi-concurrent, chapitre 4. Ces méthodes seront intégrées au système au niveau ordonnancement lors de la phase de synthèse de haut niveau.

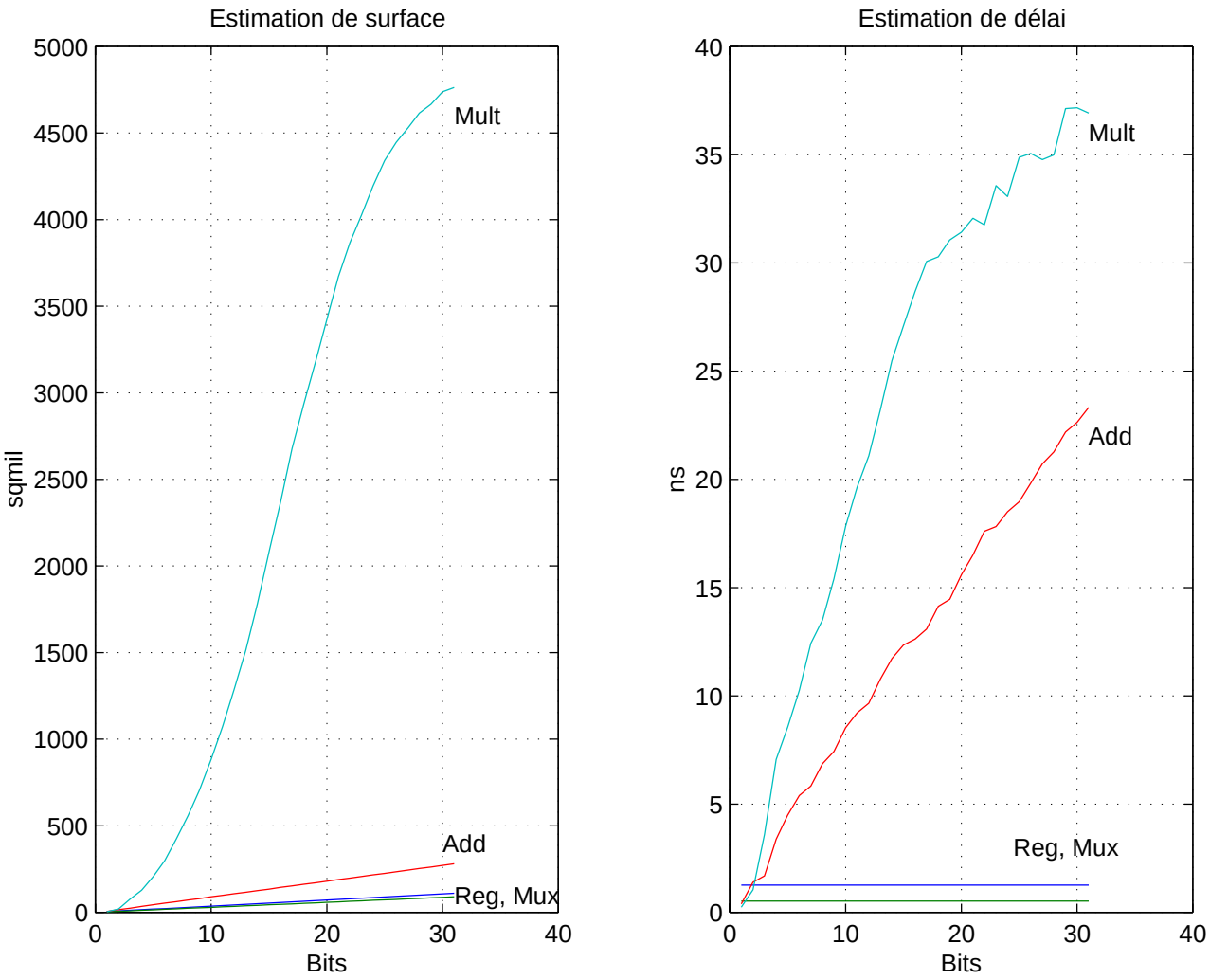


Figure 2.3: *La base de données de la cible technologique (Technologie 0.6 µm, cub de AMS).*

Partie II

Nouvelles méthodes de test en-ligne

Chapitre 3

Test en-ligne non-concurrent

La méthode de test en-ligne non-concurrent, exposée dans ce chapitre, utilise la technique de redondance temporelle au niveau unité fonctionnelle et au niveau système. Cette méthode consiste à appliquer des vecteurs de test aux unités fonctionnelles pendant leurs temps oisifs. Ces temps oisifs peuvent être identifiés dans l'intervalle fonctionnel ou dans l'intervalle oisif du système. L'intervalle fonctionnel est la période dans laquelle le système traite un échantillon, alors que l'intervalle oisif du système est la période dans laquelle le système attend l'arrivée d'un nouvel échantillon à traiter. Ces deux périodes constituent le cycle fonctionnel du système.

La testabilité en-ligne d'un système est exprimée en latence de faute. En supposant un ensemble de vecteurs de test garantissant une couverture élevée de faute, la latence de faute correspond au temps nécessaire pour l'application de l'ensemble de vecteurs de test au système.

Pour qu'un système soit testé en-ligne, les unités fonctionnelles sont supposées équipées d'un schéma de test intégré qui garanti la génération, l'application et l'analyse de la réponse d'un ensemble de vecteurs de test. Les vecteurs de test générés peuvent être déterministes ou pseudo-aléatoires. Nous considérons, dans ce chapitre, l'application de test déterministe.

Dans un cycle fonctionnel du circuit, chaque unité fonctionnelle est sollicitée pendant ses temps oisifs. Le schéma de test intégré qui lui appartient est activé pour appliquer un ou plusieurs[1] vecteurs de test et analyser la réponse. Quand il y a une opération à réaliser sur l'unité fonctionnelle, le BIST est désactivé tout en mémorisant le vecteur suivant de test à appliquer. Cela permet l'application d'un ensemble complet de vecteurs de test sur des périodes non-successives.

Ce chapitre expose, en détail, le principe de base de la méthode de test en-ligne non-concurrent et montre son introduction au niveau ordonnancement. Le coût du circuit du test en-ligne est également évalué.

3.1 Schéma de principe

Pour un système donné sous forme de graphe de flot de donnée ordonnancé (SDFG), les définitions suivantes déterminent les éléments construisant le schéma de test en-ligne :

T_{ctrl} : pas de contrôle de l'horloge fonctionnelle.

[1] Add *vs* Mult.

T_s : pas de contrôle de l'horloge d'échantillonnage qui représente le cycle fonctionnel.

T_f : intervalle fonctionnel du système, pendant lequel un échantillon est traité.

T_i : intervalle oisif du système, pendant lequel le système attend l'arrivée de l'échantillon suivant. $T_i = T_s - T_f$.

$I_{ctrl} = \frac{T_i}{T_{ctrl}}$: le nombre de pas de contrôle disponibles dans la période de repos T_i.

L_{ti} : latence de faute que permettent les temps oisifs disponibles dans la période T_i, pour l'unité fonctionnelle de type t, le temps maximal qui sépare l'apparition et la détection d'une faute dans cette unité.

L_{ts} : latence de faute que permettent les temps oisifs disponibles dans la période T_s, pour l'unité fonctionnelle de type t, le temps maximal qui sépare l'apparition et la détection d'une faute dans cette unité.

L_{faute} : latence de faute imposée au système par les contraintes de test en-ligne, le temps maximal autorisé entre l'apparition et la détection d'une faute dans le système.

V_t : nombre de vecteurs de test à appliquer sur l'unité fonctionnelle de type t.

LB_t : limite inférieure du nombre des unités fonctionnelles de type t.

NOp_{t_i} : nombre d'opérations de type t dans un pas de contrôle i.

NOs_{t_i} : nombre de temps oisifs pour l'opération de type t dans un pas de contrôle i. $NOs_{t_i} = LB_t - NOp_{t_i}$.

N_{ctrl} : nombre total de pas de contrôle dans l'intervalle fonctionnel T_f.

I_t_f : nombre de temps oisifs disponibles pour l'unité fonctionnelle de type t en T_f. $I\_t_f = \sum_{i=1}^{N_{ctrl}} NOs_{t_i}$.

I_t_s : nombre disponible de temps oisifs pour l'unité fonctionnelle de type t en T_s. $I\_t_s = I\_t_f + I_{ctrl}$.

IT_t : nombre de temps oisifs, pour l'opération de type t, nécessaire pour atteindre une latence de faute L_{faute}. Ce sont les temps oisifs de T_f qui, s'ils sont disponibles, doivent compléter les temps oisifs de la période T_i.

$$IT_t = \frac{V_t}{L_{faute}} \times T_s - I_{ctrl}$$

.

Les contraintes de testabilité en-ligne sont exprimées par la latence de faute, c'est à dire, le nombre maximal de périodes d'échantillonnage, T_s, nécessaires pour qu'une faute présente dans le système soit détectée. Etant donnée l'hypothèse que l'on a un ensemble de vecteurs de test garantissant une couverture élevée de faute, la définition de la latence de faute peut être traduite en nombre maximal de périodes d'échantillonnage nécessaires pour appliquer un test complet au système. La figure (3.1) illustre les définitions précédentes.

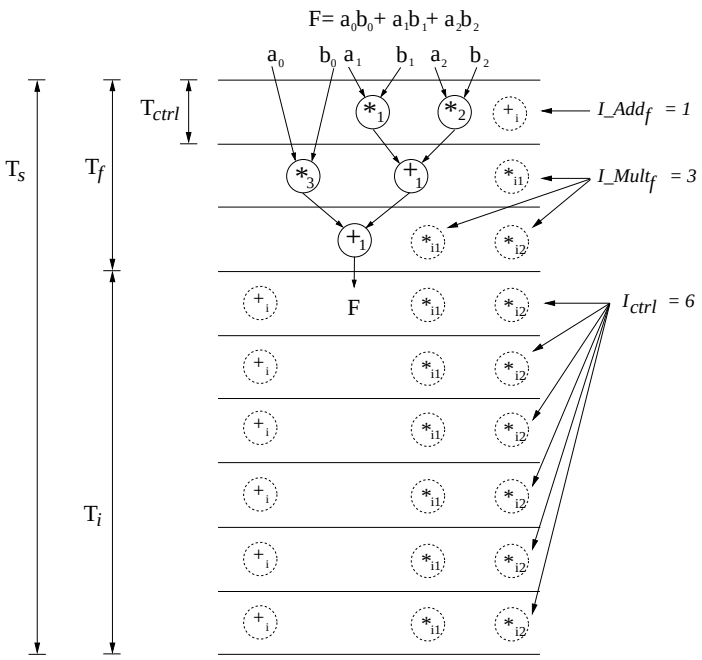


Figure 3.1: *Les éléments principaux de la méthode proposée de test en-ligne.*

3.2 Latence de faute

Nous considérons, par définition, que la latence de faute est la période maximale qui sépare l'apparition de la faute dans le système et la détection de cette faute. Pour la méthode de test en-ligne non-concurrent, les vecteurs de test, appliqués dans les temps oisifs, garantissent une couverture élevée de faute. Cela nous permet de considérer que toutes les fautes qui peuvent apparaître dans le système seront détectées par l'application de l'ensemble complet de vecteurs de test. La latence de faute correspond donc à la période dans laquelle tous les vecteurs de test peuvent être appliqués.

Pour calculer la latence de faute pour une unité fonctionnelle, il est nécessaire de disposer du nombre de temps oisifs disponible pour chaque unité fonctionnelle et du nombre de vecteurs de test à appliquer sur chaque unité fonctionnelle. Le nombre de temps oisifs dédiés au test en-ligne dans le système est représenté par I_t_f et I_{ctrl}. Le nombre de vecteurs de test pour l'unité fonctionnelle de type t existe dans une base de données contenant les paramètres des unités fonctionnelles utilisées (surface, délai, nombre de vecteurs de test, ...).

Les vecteurs de test, pour chaque unité fonctionnelle, sont appliqués dans les temps oisifs. A priori, un vecteur de test est appliqué dans un pas de contrôle. Les temps oisifs disponibles dans l'intervalle oisif T_i permettent d'atteindre une latence de faute :

$$L_{ti} = \frac{V_t}{I_{ctrl}}$$

Si cette valeur ne satisfait pas les contraintes de test en-ligne imposées au système, les temps

oisifs disponibles dans l'intervalle fonctionnel T_f sont utilisés et la latence de faute devient :

$$L_{ts} = \frac{V_t}{\frac{I\_t_f}{LB_t} + I_{ctrl}}$$

Ces valeurs indiquent le nombre de périodes d'échantillonnage nécessaires pour appliquer tous les vecteurs de test sur l'unité fonctionnelle de type t. Elles seront utilisées pour déterminer l'ordonnancement qui répond le mieux aux contraintes de test en-ligne lors de l'étape de compilation et d'ordonnancement.

3.3 Insertion des opérations de test en-ligne

Les opérations de test en-ligne sont insérées au niveau ordonnancement. Dans un premier temps, l'intervalle oisif T_i du système est identifiée. Ensuite, le nombre de temps oisifs disponibles dans cette période est calculé pour chaque unité fonctionnelle. Si ces temps oisifs sont suffisants pour atteindre la latence de faute imposée par les contraintes de test en-ligne, les opérations de test en-ligne sont insérées dans l'intervalle oisif T_i du système. Si la latence de faute n'est pas atteinte par ces temps oisifs, l'intervalle fonctionnel T_f du système est analysé pour calculer les temps oisifs disponibles. Si les temps oisifs disponibles dans les deux périodes, T_f et T_i, sont suffisants pour atteindre la latence de faute, les opérations de test en-ligne sont insérées dans les deux périodes. Dans le cas où les temps oisifs dans les deux périodes ne permettent pas d'atteindre la latence de faute, il est nécessaire de considérer une autre méthode de test en-ligne, comme les méthodes de test semi-concurrent et concurrent[2].

Considérons le cas général, dans lequel les temps oisifs disponibles dans l'intervalle oisif T_i ne permettent pas d'atteindre la latence de faute. Il est nécessaire, dans ce cas, de considérer intervalle fonctionnel T_f. L'évaluation de la testabilité en-ligne du système au niveau ordonnancement se fait par le calcul du nombre de temps oisifs disponibles en T_f. Cette étude est essentielle pour les systèmes qui ne disposent pas d'une période de repos T_i importante.

Dans un pas de contrôle i, un ou plusieurs temps oisifs existent pour l'unité fonctionnelle de type t si le nombre d'opérations de type t, NOp_t, est inférieur au nombre des unités fonctionnelles de type t au niveau RTL (LB_t). La latence de faute, que permet l'intervalle oisif T_i, est calculée. Si les contraintes de test en-ligne ne sont pas satisfaites, IT_t, le nombre de temps oisifs nécessaires pour atteindre la latence de faute imposée au système est calculé. Ce nombre manquant de temps oisifs doit être recherché dans l'intervalle fonctionnel T_f. Cette période est analysée pour savoir si ses temps oisifs correspondent à IT_t. Pour cela, le nombre des pas de contrôle dans intervalle fonctionnel N_{ctrl}, le nombre d'opérations dans chaque pas de contrôle NOp_{t_i} et le nombre des unités fonctionnelles au niveau RTL LB_t, sont évalués. Ensuite, le nombre de temps oisifs dans chaque pas de contrôle NOs_{t_i} et le nombre total de temps oisifs dans l'intervalle fonctionnel, I_t_f, sont calculés. Si $I\_t_f \geq IT_t$, les opérations de test en-ligne sont insérées dans l'intervalle fonctionnel T_f. Si $I\_t_f < IT_t$, la méthode de test en-ligne non-concurrent ne peut pas satisfaire aux contraintes de test en-ligne imposées au système et d'autres méthodes de test en-ligne doivent être explorées.

Pour illustrer le calcul de la latence de faute, l'exemple dans la figure (3.1) est utilisé. L'ordonnancement nominal de cet exemple montre le besoin d'un additionneur et de deux

[2]Perspectives : ensemble partiel (selon l'environnement) ou réorganisation des vecteurs de test.

multiplieurs au niveau RTL. Dans le tableau (3.1), tous les éléments nécessaires au calcul de la latence de faute pour l'additionneur et les multiplieurs sont présentés. On constate la disponibilité de six pas de contrôle dans l'intervalle oisif du système ($I_{ctrl} = 6$). Les temps oisifs disponibles dans la période fonctionnelle (I_{tf}) sont de 1 et 1.5 pour l'additionneur et les multiplieurs respectivement. Etant donnés le nombre d'opérateurs disponibles au niveau RTL (LB_t) et le nombre de vecteurs de test nécessaires pour chaque opérateur (V_t), la latence de faute que permet l'ordonnancement nominal peut être calculée pour l'additionneur (L_{Add}) et les multiplieurs (L_{Mult}).

t	I_{ctrl}	I_{tf}	I_{ts}	LB_t	V_t	L_{ti}	IT_t	L_{ts}	L_{faute}
Add	6	1	7	1	5	0.83	-	0.71	1
Mult	6	1.5	7.5	2	7	1.17	2	0.82	1

Table 3.1: *Evaluation de la testabilité : la latence de faute.*

L'algorithme dans la figure (3.2) évalue la testabilité des deux périodes T_i et T_s et insère les opérations de test en-ligne dans les temps oisifs d'un graphe de flot de données ordonnancé.

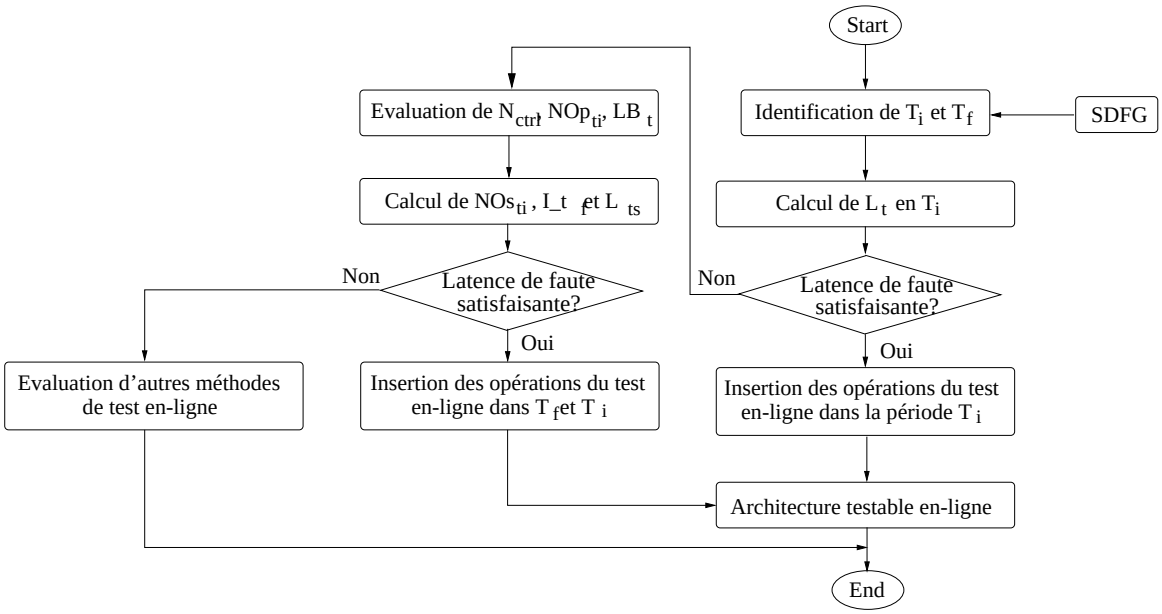


Figure 3.2: Insertion des opérations de test en-ligne non-concurrent.

3.4 L'unité de test intégré (BIST)

L'unité de test intégré (BIST) garanti l'application d'un test complet sous forme d'un ensemble prédéterminé de vecteurs de test. Ces vecteurs sont générés et appliqués dans un

ordre déterminé. A chaque pas de contrôle contenant un temps oisif, une opération de test est ordonnancée. C'est à dire qu'un ou plusieurs vecteurs de test sont appliqués à l'unité fonctionnelle et que la réponse est analysée. Lorsque l'unité fonctionnelle doit effectuer une opération nominale, la génération des vecteurs de test est instantanément bloquée. Cela permet l'application de l'ensemble de vecteurs de test de façon non successive sur les temps oisifs disponibles. Cet ensemble de vecteurs de test est appliqué de façon continue sur le système.

Le circuit de test en-ligne (BIST) se compose de deux parties, (voir figure (3.3)). La première partie contient le circuit qui assure la génération et l'application des vecteurs de test. La deuxième partie contient le circuit qui assure le codage de la réponse et l'analyse de la signature.

3.4.1 Partie génération de vecteurs de test : TG

Cette partie assure la génération et l'application d'un ensemble de vecteurs de test qui garantissent une couverture de faute élevée. Les vecteurs de test peuvent être générés par n'importe quel outil de génération de test. Nous utilisons l'outil Design_Analyzer de SYNOPSYS pour la génération des vecteurs de test dans les exemples traités dans cette étude.

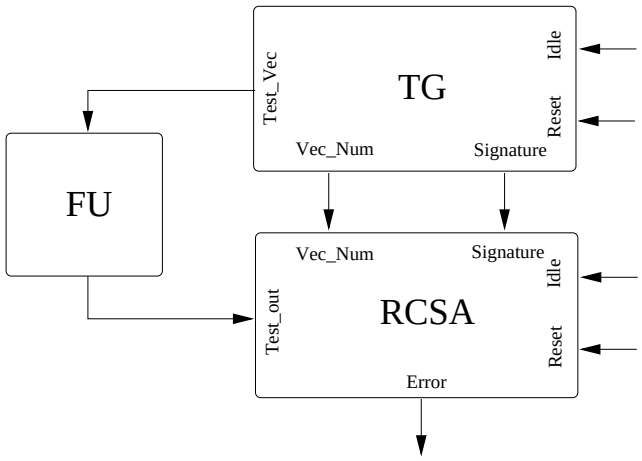


Figure 3.3: *Les deux parties du circuit de test intégré (BIST).*

L'entrée *Idle* sert à signaler au circuit l'état de repos de l'unité fonctionnelle concernée. Lorsque cette unité entre dans son temps de repos la partie génération de vecteurs de test qui lui appartient est débloquée pour permettre la génération d'un nouveau vecteur de test. En l'absence de ce signal, le circuit de génération de vecteurs de test est bloqué pour permettre la mémorisation du vecteur de test en cours. Cette partie génère aussi la signature correcte de la réponse du vecteur de test en cours. Cette signature est envoyée à la deuxième partie du BIST pour vérification de la réponse de l'unité testée. Le codage adopté pour la signature est le nombre de transitions dans la réponse.

3.4.2 Partie codage et analyse de la réponse : RCSA

Cette partie récupère la réponse de l'unité testée, génère la signature qui lui correspond, et compare la signature générée avec celle reçue de la première partie (la signature correcte). Dans le cas où les deux signatures se diffèrent, une erreur est signalée.

Elle reçoit également un signal de contrôle, *Idle*, qui lui indique que la réponse présente à la sortie de l'unité concernée est une réponse d'un vecteur de test et pas une des réponses fonctionnelles du système.

3.4.3 Evaluation du coût du BIST pour l'additionneur et le multiplieur

Le BIST qui assure le test en-ligne pour l'additionneur et pour le multiplieur a été conçu et synthétisé par l'outil design_analyzer de SYNOPSYS. Les figures(3.5) et (3.6) représentent une évaluation du coût en surface et du délai pour le circuit de test par rapport aux additionneurs et multiplieurs. Les unités fonctionnelles et les unités de test en-ligne ont été décrites en VHDL comportemental et synthétisées avec l'outil design_analyzer de SYNOPSYS. Ensuite, le coût en surface et en délai a été donné, par l'outil, pour chaque largeur en bit. On constate que le coût en surface des deux partie du BIST est bien comparable à celui d'un additionneur. Alors que le coût en délai reste nettement inférieur à celui de l'additionneur.

3.5 Conclusion

Dans ce chapitre, la méthode de test en-ligne non-concurrent a été exposée. Un circuit de test en-ligne a été conçu et son coût en surface et délai a été évalué. L'intégration de cette méthode de test en-ligne dans le flot général de HLS_OLT proposé par cette étude est présentée dans la figure (3.4).

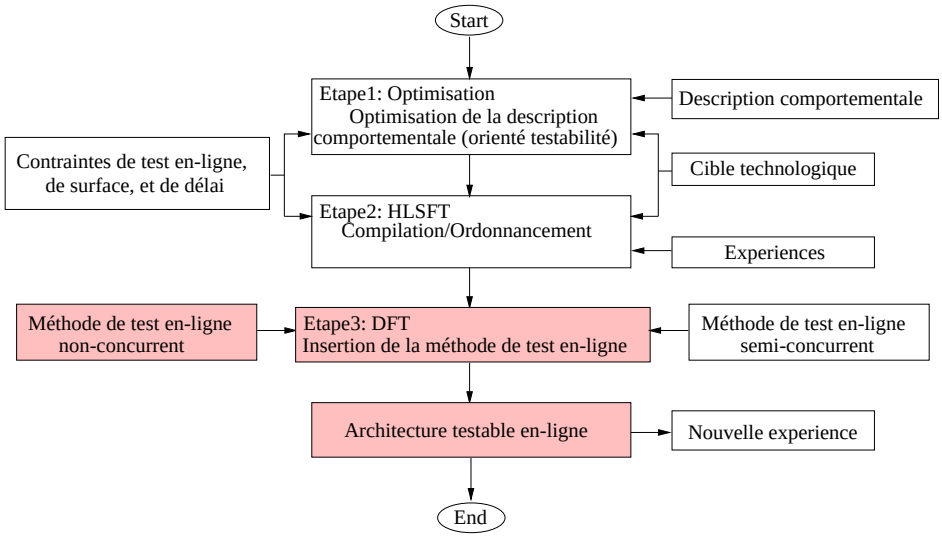


Figure 3.4: *L'intégration de la méthode de test en-ligne non-concurrent dans le flot de la synthèse de haut niveau pour le test en-ligne (HLS_OLT).*

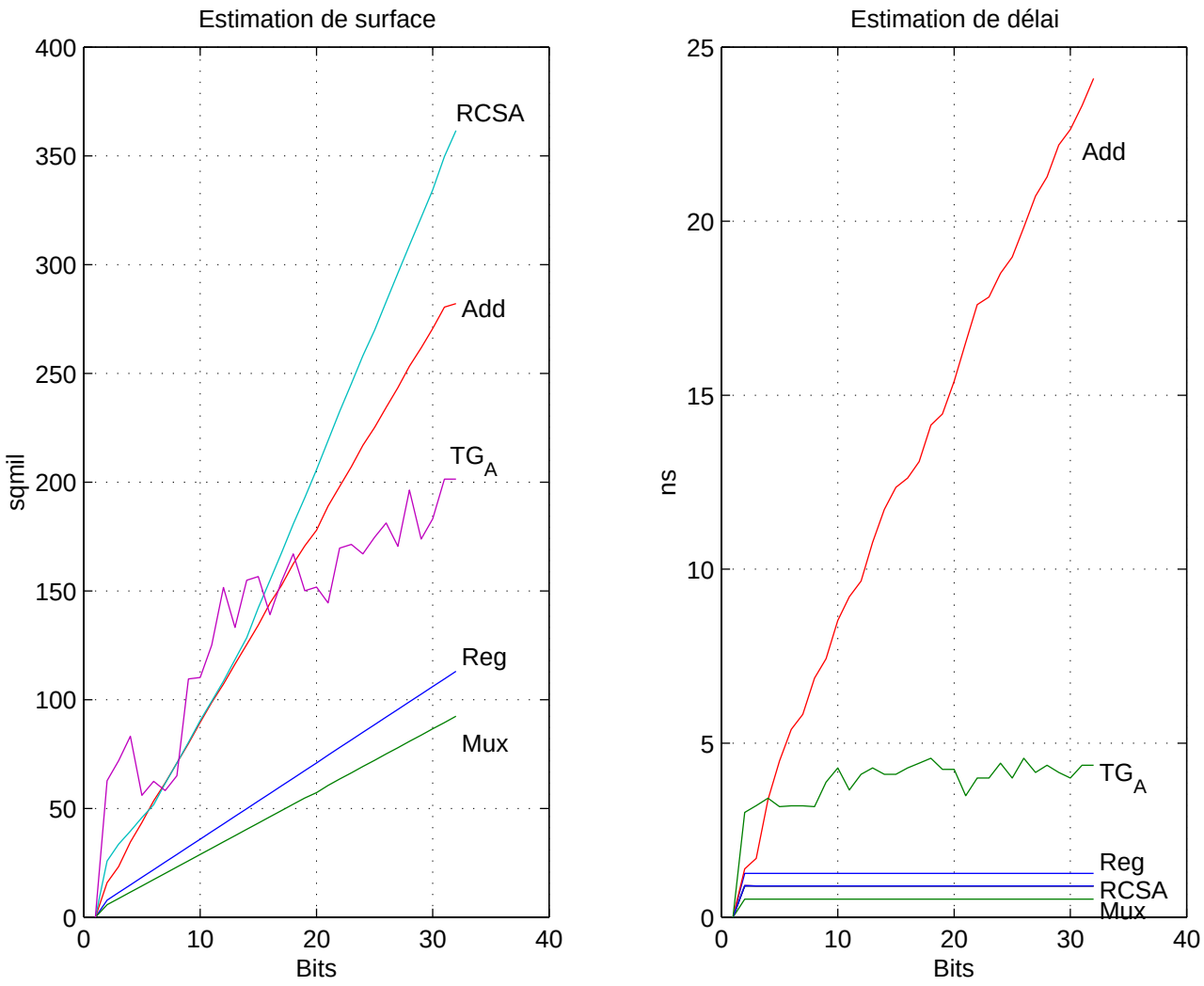


Figure 3.5: *Le circuit de test intégré pour l'additionneur (Technologie 0.6 μm, cub de AMS).*

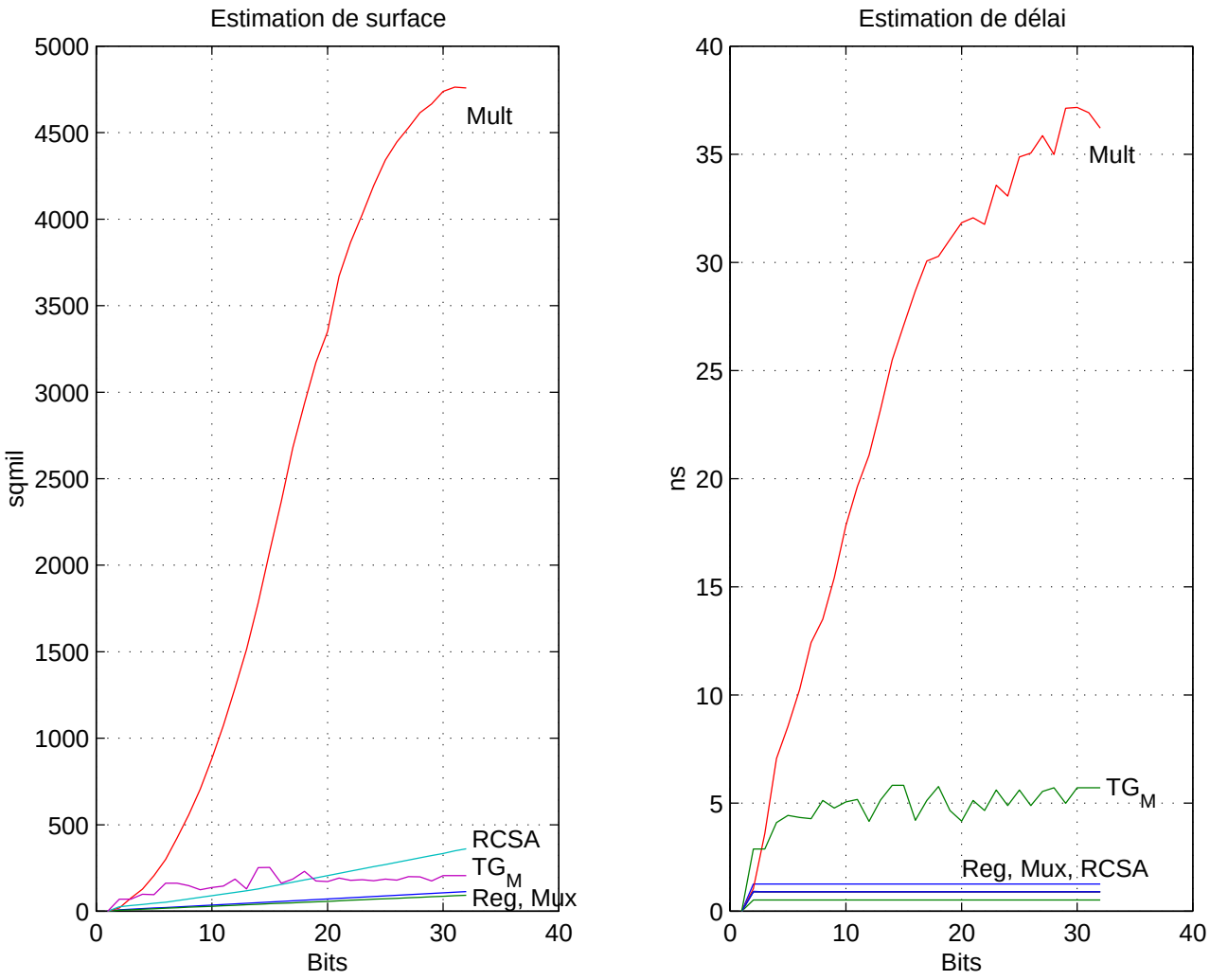


Figure 3.6: *Le circuit de test intégré pour le multiplieur (Technologie 0.6 μm, cub de AMS).*

Chapitre 4

Test en-ligne semi-concurrent

La méthode de test en-ligne non-concurrent, exposée dans le chapitre précédent, est bien adaptée aux systèmes disposant d'un intervalle oisif, T_i dans la figure (3.1), suffisant pour atteindre une latence de faute déterminée. Pour les systèmes qui ne disposent pas d'un tel intervalle oisif ou dont cet intervalle est insuffisant, la méthode de test en-ligne non-concurrent ne convient pas. Il est donc nécessaire de développer d'autres méthodes de test en-ligne. Dans ce chapitre, l'accent est mis sur le développement de méthodes de test en-ligne qui appliquent un test fonctionnel sur le système. Ces méthodes utilisent aussi la redondance temporelle dans le système comme le fait la méthode de test en-ligne non-concurrent. La différence entre ces deux types de test en-ligne est que les méthodes proposées ici ne visent pas à appliquer des vecteurs de test dans le sens de générations de test. Les données nominales appliquées aux entrées principales du système sous test sont la seule excitation utilisée pour effectuer le test en-ligne. Le flot de données sur les entrées du système peut être considéré comme une séquence très longue de vecteurs de test aléatoires permettant ainsi une couverture de faute acceptable. Ces méthodes sont considérées comme des méthodes de test semi-concurrent en raison de l'utilisation des entrées fonctionnelles du système, comme le font les méthodes de test concurrent, et de l'application du test pendant l'intervalle oisif des unités fonctionnelles, comme le font les méthodes de test non-concurrent.

Ce type de méthodes convient aux systèmes qui fonctionnent en continu et qui disposent de façon permanente d'entrées fonctionnelles. Pour cette raison, le test en-ligne semi-concurrent ne peut pas exploiter efficacement l'intervalle oisif du système (la période T_i) si elle existe. Ceci est dû à la dépendance de ce type de tests des valeurs fonctionnelles des entrées de l'opérateur. Si ces valeurs sont absentes ou figées le test est pratiquement inefficace.

Dans ce chapitre, des nouvelles méthodes de test en-ligne semi-concurrent sont exposées. Ces méthodes résultent de deux principes. Le premier principe consiste à appliquer un test fonctionnel par l'exploitation de la distributivité de la multiplication sur l'addition. Le deuxième principe consiste à refaire le calcul nominal qui vient d'être effectué par le système ou par une partie de ses composants. Ces deux principes peuvent être appliqués localement au niveau de chaque unité fonctionnelle ou globalement pour vérifier l'intégralité du calcul effectué par le système nominal.

La redondance temporelle est exploitée pour appliquer ces méthodes de test en-ligne semi-concurrent. Les unités fonctionnelles sont réutilisées dans leurs temps oisifs pour le

test en-ligne. Le deuxième principe, refaire le calcul nominal, nécessite aussi une redondance matérielle. Nous entendons par une redondance matérielle le fait de disposer de plusieures unités fonctionnelles du même type dans la structure RTL du système. En exploitant une telle redondance matérielle une permutation d'unités fonctionnelles est réalisée pour le test en-ligne. Les opérations de vérification de calcul sont réalisées sur d'autres unités fonctionnelles que celles des opérations nominales. Ce type de redondance matérielle n'est pas nécessaire pour le premier principe (l'exploitation de la distributivité).

4.1 Exploitation de la distributivité

Nous allons exposer le schéma de principe de l'exploitation de la distributivité puis son application au niveau local et global du système. Cette méthode engendre l'augmentation de la largeur en bit des unités fonctionnelles pour ne pas perdre une partie de l'entrée par le décalage. Pour cette raison, le coût en surface et en délai de cette méthode est évalué.

4.1.1 Schéma de principe

Cette méthode consiste à vérifier le calcul qui a été effectué sur l'unité fonctionnelle en utilisant la propriété de la distributivité de la multiplication sur l'addition. Pour cela, les entrées de l'unité considérée sont multipliées par 2^i, $i = 1, 2, \ldots$ est choisi selon l'objectif du test. Les entrées décalées sont appliquées de nouveau sur l'unité fonctionnelle. Le résultat nominal, qui a été obtenu à partir des entrées normales, est multiplié aussi par 2^i, pour l'addition, ou par 2^{2i} pour la multiplication, et comparé avec le résultat des entrées décalées. En cas d'inégalité entre les deux résultats, une erreur est signalée.

Considérons un additionneur dans une structure RTL pendant deux pas de contrôle. Dans le premier pas de contrôle, l'additionneur est fonctionnel. Deux valeurs sont présentes à ses entrées. Dans le deuxième pas de contrôle, l'additionneur entre dans un temps oisif. Les valeurs qui étaient présentes sur ses entrées dans le pas de contrôle précédent sont décalées et réutilisées dans ce temps oisif. La sortie de l'intervalle fonctionnel est aussi décalée et préparée pour la comparaison avec la sortie du temps oisif. La figure (4.1) représente l'additionneur dans ces deux pas de contrôle.

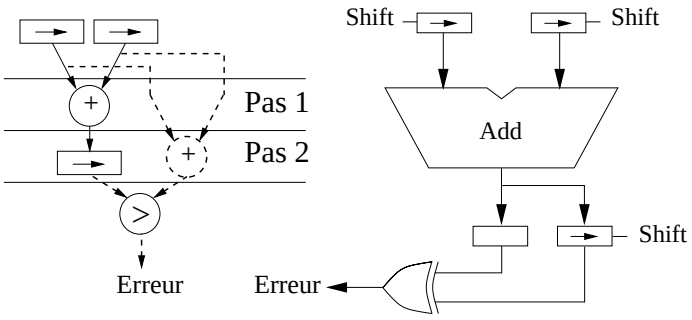


Figure 4.1: *Exploitation de la distributivité au niveau unité fonctionnelle.*

4.1.2 Test global

Cette méthode peut être utilisée pour vérifier chaque unité fonctionnelle de façon indépendante des autres unités dans le système. Il est cependant facile de généraliser cette méthode pour une vérification globale du système. Soit un système de calcul qui réalise l'équation suivante :

$$y = A(B + C) + D$$

Le graphe de flot de données ordonnancé qui représente cette équation, figure (4.2), peut être réalisé, au niveau RTL, par un additionneur et un multiplieur.

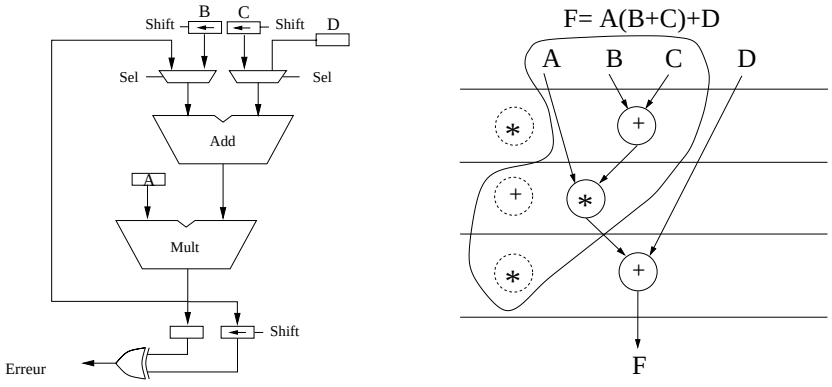


Figure 4.2: *Exploitation globale de la distributivité.*

Pour une vérification globale du système, le test fonctionnel qui sera appliqué vérifie l'égalité :

$$A(2B + 2C) = 2A(B + C)$$

Ce test vérifie le bon fonctionnement de l'additionneur et du multiplieur et donne le résultat avec la fin du calcul nominal. La réalisation du test consiste à décaler les registres contenant les variables B et C d'un bit à gauche. Les valeurs décalées de B et C sont introduites dans l'additionneur dans sa période de repos et le résultat de $2B + 2C$ est multiplié par A dans l'intervalle oisif du multiplieur. Le résultat nominal de $A(B + C)$ est décalé à gauche d'un bit et comparé avec le résultat de $A(2B + 2C)$. Les étapes suivantes expliquent en détails l'application du test en-ligne pour cet exemple :

Pas1 : les valeurs nominales de B et C sont appliquées aux entrées de l'additionneur. Le multiplieur est en temps oisif mais cette période ne peut pas être utilisée en raison de l'absence des entrées fonctionnelles.

Pas2 : avec l'horloge, la sortie de l'additionneur, $B + C$, passe à une des entrées du multiplieur et les valeurs de B et de C sont décalées à gauche d'un bit. L'additionneur, étant dans ce pas de contrôle en temps oisif, est utilisé pour calculer $2B + 2C$.

Pas3 : le résultat de la multiplication $A(B + C)$ est prêt. Il est sauvegardé dans un registre. Le résultat de $2B + 2C$ est aussi prêt à la sortie de l'additionneur et est passé au multiplieur. Celui-ci entre en temps oisif. Il est donc utilisé pour calculer $A(2B+2C)$. A la fin de ce pas de contrôle, le résultat de $A(2B+2C)$ est prêt à la sortie du multiplieur. Le résultat de $A(B + C)$, qui a été sauvegardé dans un registre, est décalé à gauche d'un bit pour produire $2A(B + C)$. Les deux résultats sont comparés et, s'ils sont différents, une erreur est signalée.

Avec cette généralisation, jusqu'à 50% de la surface supplémentaire dû au circuit de test peut être économisée. Au lieu de réaliser deux circuits de test en-ligne, un pour l'additionneur et un pour le multiplieur, un seul circuit a pu être utilisé pour le test des deux unités. Il est nécessaire, cependant, d'identifier les blocs d'opérations qui permettent le test complet du chemin de données. Ces blocs peuvent exister, mais pas nécessairement, dans le chemin critique du système. En fait, le choix des blocs d'opérations pour ce type de test en-ligne revient au choix des variables communes qui représentent le meilleur gain pour la factorisation. L'identification de telles variables sera adressée dans le chapitre 5. A noter que le chemin identifié pour ce type de test en-ligne doit contenir une seule opération de multiplication pour éviter la dégradation en surface qu'implique l'utilisation de plus larges multiplieurs.

4.1.3 Ordonnancement des opérations de test en-ligne

Pour effectuer une opération de test en-ligne il faut disposer de deux éléments. Le premier élément est la disponibilité d'entrées fonctionnelles qui auriont été déja utilisées pour réaliser une opération nominale. Le deuxième élément est la disponibilité des temps oisifs nécessaires pour effectuer les opérations de test en-ligne. Toute opération effectuée dans l'ordonnancement nominale est stockée dans une liste d'attente. Dans un pas de contrôle suivant et lorsque un temps oisif, correspondant au type de cette opération, est disponible une opération de test en-ligne est ordonnancée. A la fin de l'opération de test, le résultat nominal est comparé avec celui de test et une éventuelle erreur est signalée. L'algorithme dans la figure (4.3) montre l'ordonnancement des opérations de test dans l'ordonnancement nominal du système.

4.1.4 Exploitation de la redondance temporelle et matérielle

Si la structure RTL du système dispose de plusieures unités fonctionnelles du même type, la permutation des unités fonctionnelles pour l'exécution des opérations de test en-ligne augmente l'efficacité du test et diminue son coût. Dans ce cas, la redondance matérielle est exploitée avec celle temporelle pour effectuer le test en-ligne. Cette méthode peut être illustrée par l'exemple dans la figure (4.4).

Comme le montre la figure (4.2), le test vérifie l'égalité :

$$A(2B + 2C) = 2A(B + C)$$

Cette égalité est vérifiée deux fois dans cette méthode. Une fois par le chemin $Add_1 : Mult_1$ et une autre fois par le chemin $Add_2 : Mult_2$. Dans ce cas, en décalant seulement la moitié des entrées fonctionnelles du système nous arrivons à réaliser un test de toutes les unités fonctionnelles de sa structure RTL.

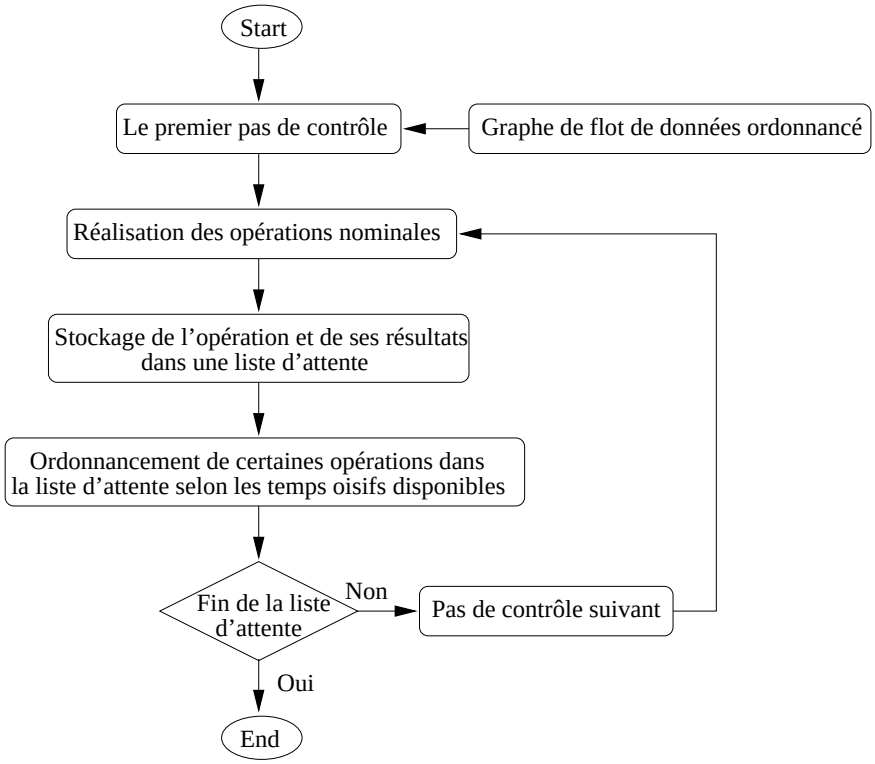
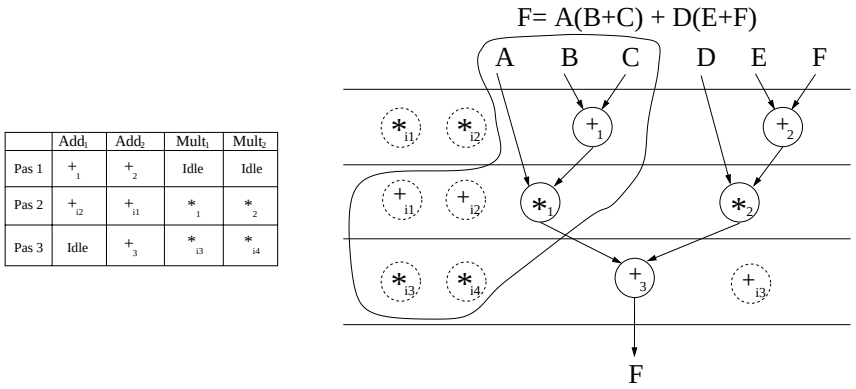


Figure 4.3: *Ordonnancement des opérations de test en-ligne.*



	Add_1	Add_2	$Mult_1$	$Mult_2$
Pas 1	$+_1$	$+_2$	Idle	Idle
Pas 2	$+_{i2}$	$+_{i1}$	$*_1$	$*_2$
Pas 3	Idle	$+_3$	$*_{i3}$	$*_{i4}$

Figure 4.4: *Exploitation globale de la distributivité avec permutation d'unités fonctionnelles.*

Pour implémenter cette méthode, l'algorithme dans la figure (4.3) doit tenir compte de la redondance matérielle. Dans ce cas, l'opération de test en-ligne doit être effectuée sur toutes les unités fonctionnelles disponibles dans le pas de contrôle considéré.

Bien que ce type de méthodes soitent très raisonnables en surface et délai supplémentaires, elles nécessitent que l'opérateur visé par le test soit fonctionnel au moins dans un pas de contrôle avant d'activer le test. Les entrées et sorties fonctionnelles sont nécessaires pour l'obtention et la vérification du résultat du test. Ceci implique que certains temps oisifs au début de l'intervalle fonctionnel ne peuvent pas être utilisés pour ce type de test. Cela peut affecter le gain en temps oisif qui peut être obtenu par l'optimisation de la description comportementale.

4.1.5 Circuit de test en-ligne

L'implémentation de la méthode de test en-ligne basé sur l'exploitation de la distributivité implique plusieurs modifications sur l'architecture nominale du système. Pour une multiplication des entrées d'une unité fonctionnelle par 2^i, il est nécessaire de remplacer l'unité nominale et ses registres d'entrées/sorties de n bits par une autre de $n+i$ bits. En plus, les registres des entrées/sorties sont transformés en registres à décalage. Un registre à décalage supplémentaire est ajouté pour sauvegarder le résultat nominal en vue de le comparer avec le résultat du test.

Le schéma dans la figure (4.5) montre la réalisation du multiplieur testable en-ligne. La multiplication et la division par 2, ($i = 1$), des entrées du multiplieur sont illustrées.

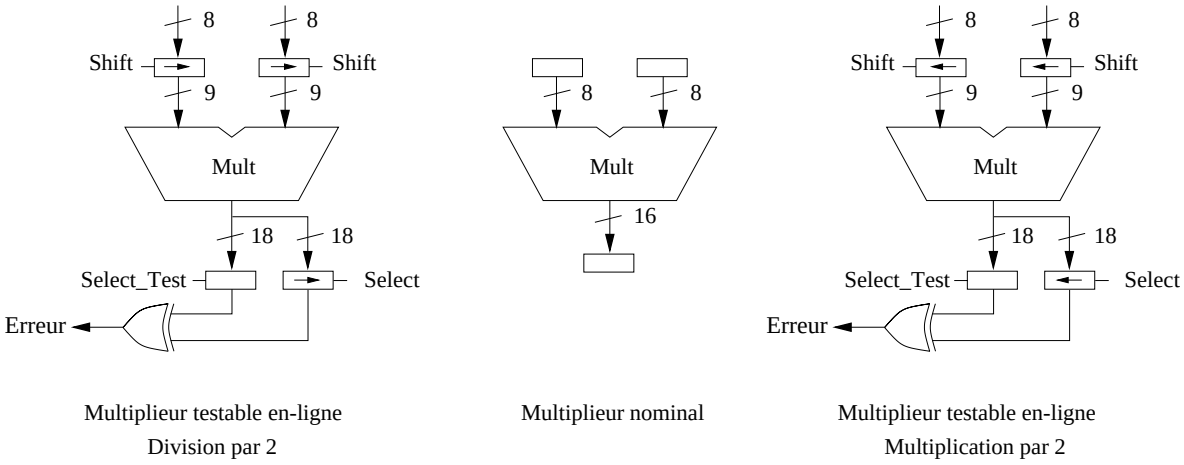


Figure 4.5: *Exploitation de la distributivité: réalisation du circuit avec* $i = 1$.

Pour comprendre le fonctionnement de l'architecture testable en-ligne au niveau RTL, considérons une multiplication par 2 des entrées d'un multiplieur 8 bits, figure (4.5). La sortie nominale doit être multipliée par 4 pour pouvoir comparer le résultat nominal avec le résultat de test en-ligne. Le multiplieur testable en-ligne est de 9 bits. Les registres des entrées/sortie deviennent également de 9 bits. Les données nominales de 8 bits sont connectées aux premiers 8 bits des registres des entrées. A la fin de l'intervalle fonctionnel,

le résultat nominal est sauvegardé dans le registre à décalage. Dans le temps oisif, le contenu des registres des entrées est décalé d'un bit à gauche et appliqué sur le multiplieur 9 bits. Le résultat de cette opération est sauvegardé dans le registre de test. Le résultat nominal, stocké dans le registre à décalage, est décalé de deux bits et comparé avec le résultat du test. Pour $i = 1$, le surcoût, en surface et en délai, de la méthode est présenté dans la figure (4.6).

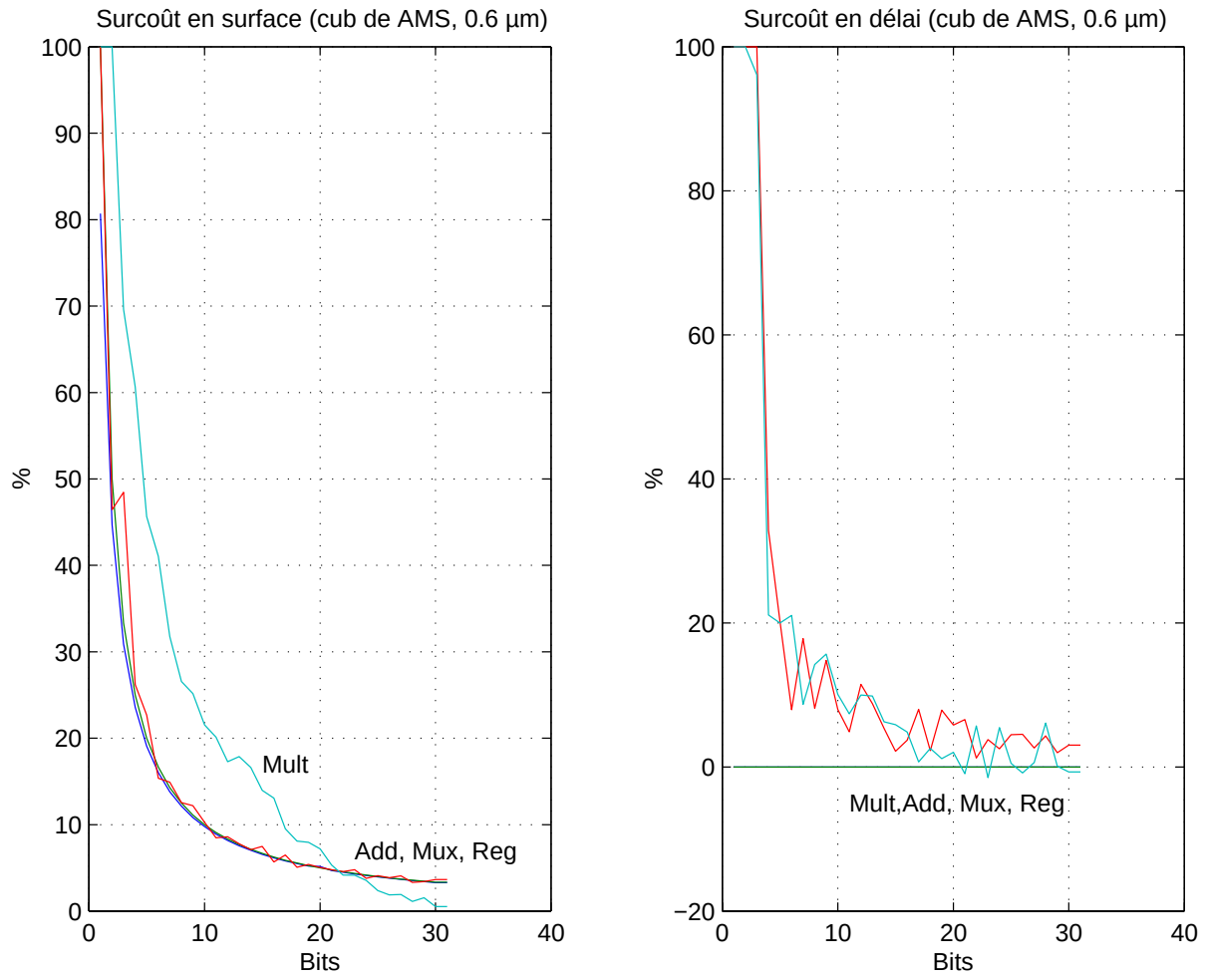


Figure 4.6: *Exploitation de la distributivité: surcoût en surface et en délai des différentes unités.*

Chaque élément, de largeur l, dans l'architecture RTL sera remplacé par un autre de largeur $l + 1$. La différence, en surface et en délai, entre les deux largeurs en bit est calculée à partir des informations concernant la surface et le délai de chaque élément pour chaque largeur en bit, (figure (3.6)).

On peut constater que le coût de la méthode, en surface et en délai, varie de 12% jusqu'à 30% pour des architectures basées sur des opérateurs 8 bits. Cependant ce coût devient très

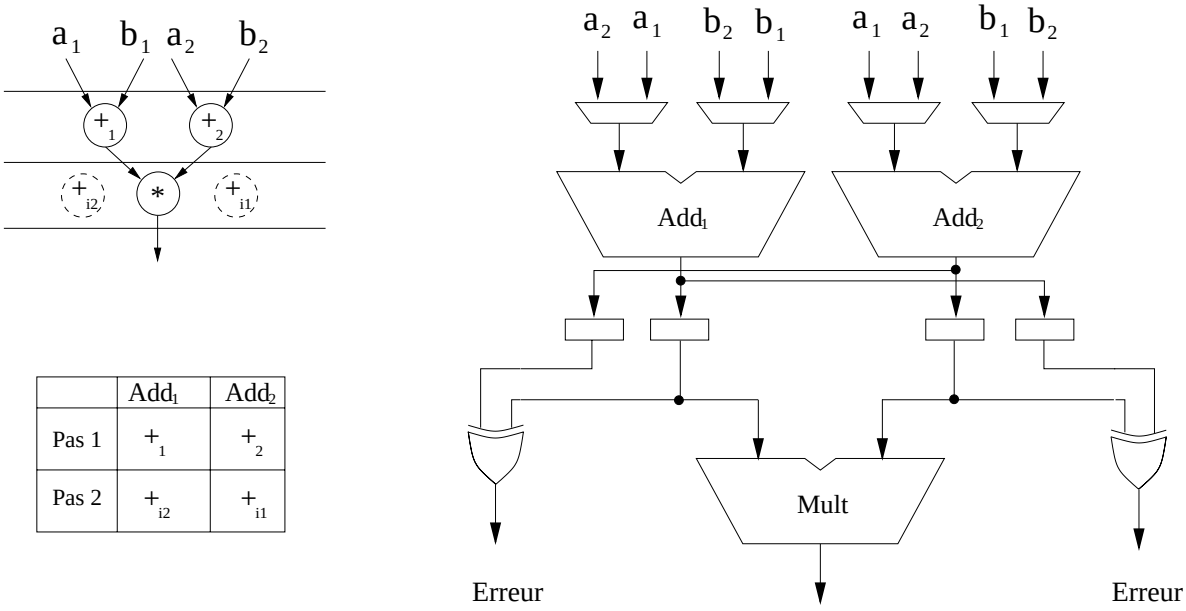
raisonnable pour les architectures complexes utilisant des opérateurs 16 bits et plus.

4.2 Vérification de calcul

Cette méthode a pour objectif de vérifier le calcul effectué par le système. Les unités fonctionnelles sont réutilisées dans leurs temps oisifs pour refaire le calcul nominal. Les deux résultats sont comparés pour indiquer le bon fonctionnement du système. Cette méthode permet de vérifier l'exactitude du résultat produit par le système nominal. Elle peut être réalisée localement pour chaque unité fonctionnelle ou globalement pour le système entier.

4.2.1 Schéma de principe

Si la structure RTL du système dispose de plusieures unités fonctionnelles, il est possible de refaire le calcul nominal qui vient d'être effectué en permutant les unités fonctionnelles. L'opération de test est alors réalisée sur une unité fonctionnelle autre que celle de l'opération nominale. Pour illustrer cette méthode considérons l'ordonnancement dans la figure (4.7).



	Add$_1$	Add$_2$
Pas 1	$+_1$	$+_2$
Pas 2	$+_{i2}$	$+_{i1}$

Figure 4.7: *Vérification locale du calcul avec permutation d'unités fonctionnelles.*

Dans un premier temps, l'additionneur Add_1 réalise l'opération $a_1 + b_1$ et l'additionneur Add_2 réalise l'opération $a_2 + b_2$. Dans un deuxième temps, les deux additionneur sont oisifs et ils peuvent être utilisés pour vérifier ces deux opérations. pour cela l'opération $a_1 + b_1$ est refaite sur l'additionneur Add_2 alors que l'opération $a_2 + b_2$ est refaite sur l'additionneur Add_1 et les résultats de test sont comparés avec les résultats nominaux.

L'algorithme d'ordonnancement des opérations de test en-ligne dans la figure (4.3) peut être utilisé pour cette méthode. Une condition est, cependant, à imposer lors de l'ordonnancement d'une opération de test en-ligne dans un temps oisif. Cette condition consiste à vérifier la

permutation des unités fonctionnelles pour les opérations de test en-ligne. Celle-ci doit être allouée à une unité fonctionnelle autre que celle de l'opération nominale.

4.2.2 Test global

Pendant les temps oisifs des unités fonctionnelles, il est possible d'ordonnancer la séquence d'opérations du graphe de flot de données nominal. Le graphe secondaire servira à reproduire le résultat nominal du système pour le comparer avec le résultat du graphe nominal. Cette idée convient au test en-ligne des fautes intermittentes. Une petite modification rend ce test convenable aussi aux fautes permanentes. Cette modification consiste à permuter les unités fonctionnelles, lors de l'allocation de ressources, pour exécuter les opérations du graphe secondaire (graphe de test) sur des unités fonctionnelles différentes de celles des opérations du graphe nominal. Cela sert, par ailleurs, à minimiser le risque de masquage de fautes. Les définitions suivantes sont nécessaires pour établir cette étude :

T_{ctrl} : pas de contrôle de l'horloge fonctionnelle.

N_{ctrl} : nombre total de pas de contrôle du système nominal.

$L_N = N_{ctrl}$: la latence du système nominal.

L_t : latence de faute que permettent les temps oisifs disponibles.

L_F : la latence de faute que doit atteindre le test en-ligne.

L'architecture nominale est la structure RTL qui implémente le graphe nominal en tenant compte seulement des contraintes de coût et de performance. Aucune contrainte de testabilité n'est considérée dans cette architecture.

Le graphe nominal est réutilisé pour générer, au niveau ordonnancement, l'architecture du test en-ligne. Pour cela, ce graphe est ordonnancé de nouveau dans les pas de contrôle de l'architecture nominale. Sans violer les contraintes de coût et de performance, cet ordonnancement doit satisfaire à une latence de faute L_F. Le résultat est une architecture auto-testable en-ligne composée d'un graphe nominal qui sert à produire le résultat nominal à une latence L_N et du même graphe pour reproduire le même résultat mais à une latence $L_t \geq L_F$. Cette architecture, auto-testable en-ligne, fonctionne de la manière suivante :

- au début du cycle de vérification, les deux architectures reçoivent les données nominales du système et les traitent de façon indépendante.
- après L_N pas de contrôle, l'architecture nominale délivre le résultat nominal du cycle opérationnel, ce résultat est stocké dans un registre en attendant le résultat de l'architecture de test en-ligne. L'architecture nominale reçoit, de nouveau, des données à traiter alors que l'architecture de test en-ligne continue le calcul du résultat des données précédentes.
- après L_F pas de contrôle, l'architecture de test en-ligne délivre son résultat, qui doit être, en principe, identique au résultat nominal qui a été délivré et stocké par l'architecture nominale. Les deux résultats sont comparés. Si les deux résultats sont identiques, un nouveau cycle de vérification peut commencer. Sinon, un signal d'erreur est activé.

A priori, les opérations de l'architecture du test en-ligne sont toutes ordonnancées dans les temps oisifs de l'architecture nominale. Cela signifie que les deux architectures partagent de façon complète les unités fonctionnelles au niveau RTL. Cependant, la latence de faute L_F peut exiger un ordonnancement plus concentré des opérations de test en-ligne. Cela peut impliquer l'introduction des unités fonctionnelles supplémentaires pour répondre à ce besoin. Si le coût total en surface ne viole pas les contraintes imposées au système, l'architecture globale est réalisée.

Pour illustrer le principe et l'implémentation de cette méthode, considérons le système représenté par son graphe de flot de données ordonnancé dans la figure (4.8)

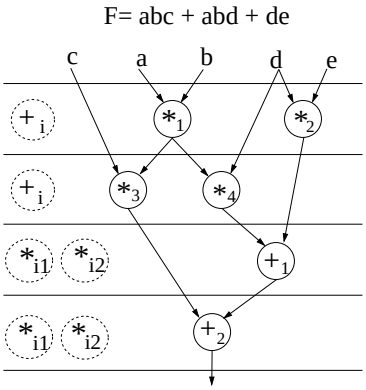


Figure 4.8: *L'ordonnancement de l'architecture nominale.*

Pour générer l'architecture de test en-ligne, le graphe de flot de données est ordonnancé de nouveau dans les temps oisifs de l'ordonnancement nominal. L'ordonnancement de l'architecture testable en-ligne est présenté dans la figure (4.9).

Le tableau (4.1) montre l'allocation des unités fonctionnelles et l'assignation des opérations de l'architecture testable en-ligne. A noter que la permutation des unités fonctionnelles dans l'architecture du test en-ligne peut se faire quand il y a plusieurs unités fonctionnelles. C'est le cas des multiplieurs dans cet exemple. Pour l'addition, il n'y a qu'un seul additionneur au niveau RTL.

Pas de contrôle	$Mult_1$	$Mult_2$	Add
1	$*_1$	$*_2$	$+_{c1}$
2	$*_3$	$*_4$	$+_{c2}$
3	$*_{c2}$	$*_{c1}$	$+_1$
4	$*_{c4}$	$*_{c3}$	$+_2$

Table 4.1: *L'allocation de ressource et l'assignation de l'architecture testable en-ligne.*

Pour contourner la difficulté du test en-ligne de l'additionneur et pour améliorer la qualité du test en-ligne, il est possible de combiner cette méthode avec la méthode basée sur

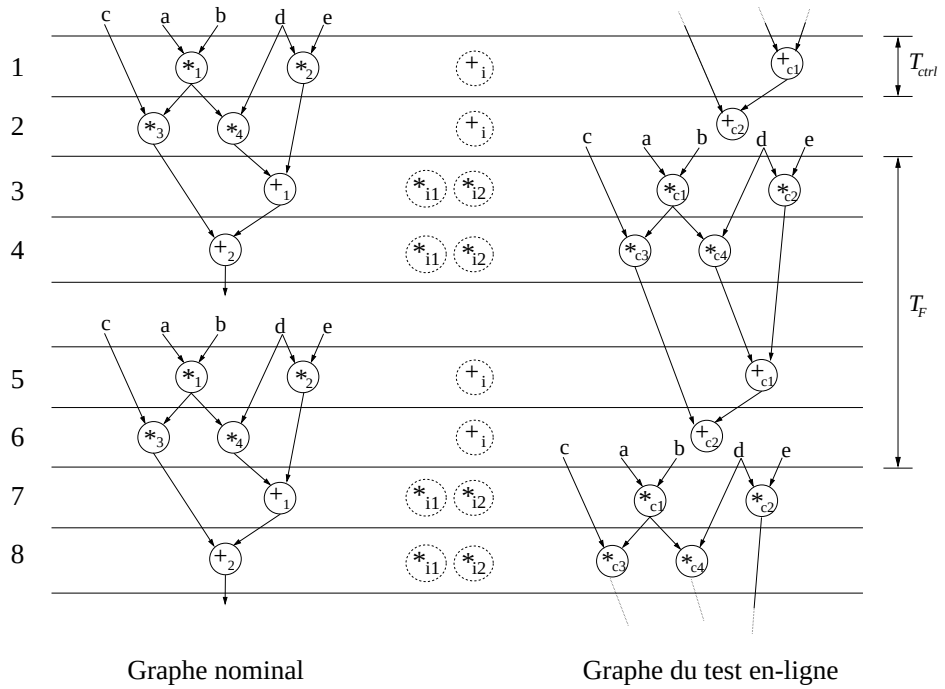


Figure 4.9: *L'ordonnancement de l'architecture testable en-ligne.*

l'exploitation de la distributivité. Dans ce cas, les données nominales sont décalées avant d'être appliquées au graphe de test en-ligne. Ainsi, le résultat nominal est décalé avant la comparaison avec le résultat du graphe de test en-ligne.

4.2.3 Exploitation des variante-duales de DFGs

La méthode de la vérification globale du calcul, présentée dans la section précédente, peut présenter un problème important pour certains types de système. Cela concerne les systèmes dont l'architecture du graphe de flot de données ne convient pas à la distribution des temps oisifs dans les pas de contrôle de l'ordonnancement nominal. Dans ce cas, soit l'ordonnancement de l'architecture du test en-ligne imposera l'introduction de nouveaux matériels au niveau RTL, soit la latence de faute doit être large. Pour illustrer ce problème, considérons l'exemple de l'équation différentielle :

$$\begin{aligned} u^{'} &= u - 3xudx - 3ydx \\ y^{'} &= y + udx \\ x^{'} &= x + dx \end{aligned}$$

La figure (4.10) représente l'ordonnancement de l'architecture nominale du système. La disponibilité des temps oisifs dans les pas de contrôle est représentée par des opérations en pointillés.

L'application de la méthode de la vérification globale du calcul implique l'ordonnancement du graphe de flot de données dans les temps oisifs. Considérons le cas où les contraintes de

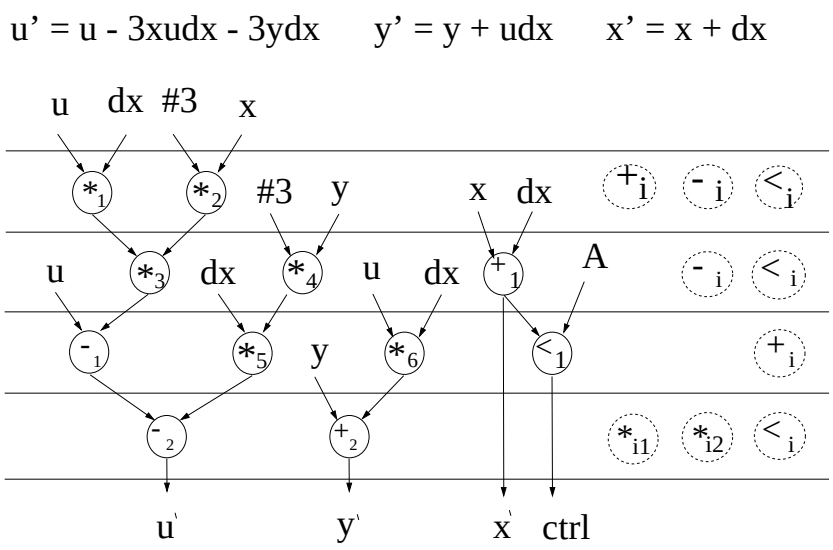


Figure 4.10: *L'ordonnancement nominal de l'équation différentielle.*

surface ne permettent pas l'ajout d'unités fonctionnelles supplémentaires. L'ordonnancement du graphe nominal dans les temps oisifs s'effectue comme le montre la figure (4.11). Cet ordonnancement du graphe de test en-ligne permet une latence de faute $L_F \geq 12$ pas de contrôle.

Ce résultat peut être nettement amélioré par l'exploitation d'une variante-duale du graphe de flot de données de la première équation du système.

Cette équation peut être factorisée, elle devient : $u' = u - 3(xudx - ydx)$. Cela donne, au niveau ordonnancement, le graphe de la figure (4.12).

L'ordonnancement de ce nouveau graphe dans les temps oisifs du système nominal dans la figure (4.10) donne le système testable en-ligne de la figure (4.13). La latence de faute, $L_F \geq 8$ pas de contrôle dans ce cas, est améliorée de 33%. L'allocation de ressources de l'architecture testable en-ligne avec le graphe nominal et la variante-duale est donnée dans le tableau (4.2).

4.2.4 Latence de faute

L'efficacité du test en-ligne dépend de la structure du graphe de test en-ligne et de la distribution des temps oisifs dans l'ordonnancement nominal. La testabilité en-ligne d'un système correspond au nombre de cycles fonctionnels nécessaires pour l'ordonnancement du graphe de test en-ligne. Ce nombre peut être estimé par l'ordonnancement des opérations du chemin critique.

Pour un système ayant plusieurs variante-duales de graphe de flot de données, le choix du graphe de test en-ligne se fait après une évaluation de la latence de faute de chaque variante-duale. Pour ce faire, le chemin critique est identifié et une liste de ses opérations est établie pour chacune des variante-duales. Le nombre de cycles fonctionnels nécessaires pour ordonnancer la liste des opérations du chemin critique est calculé. Ce nombre représente la

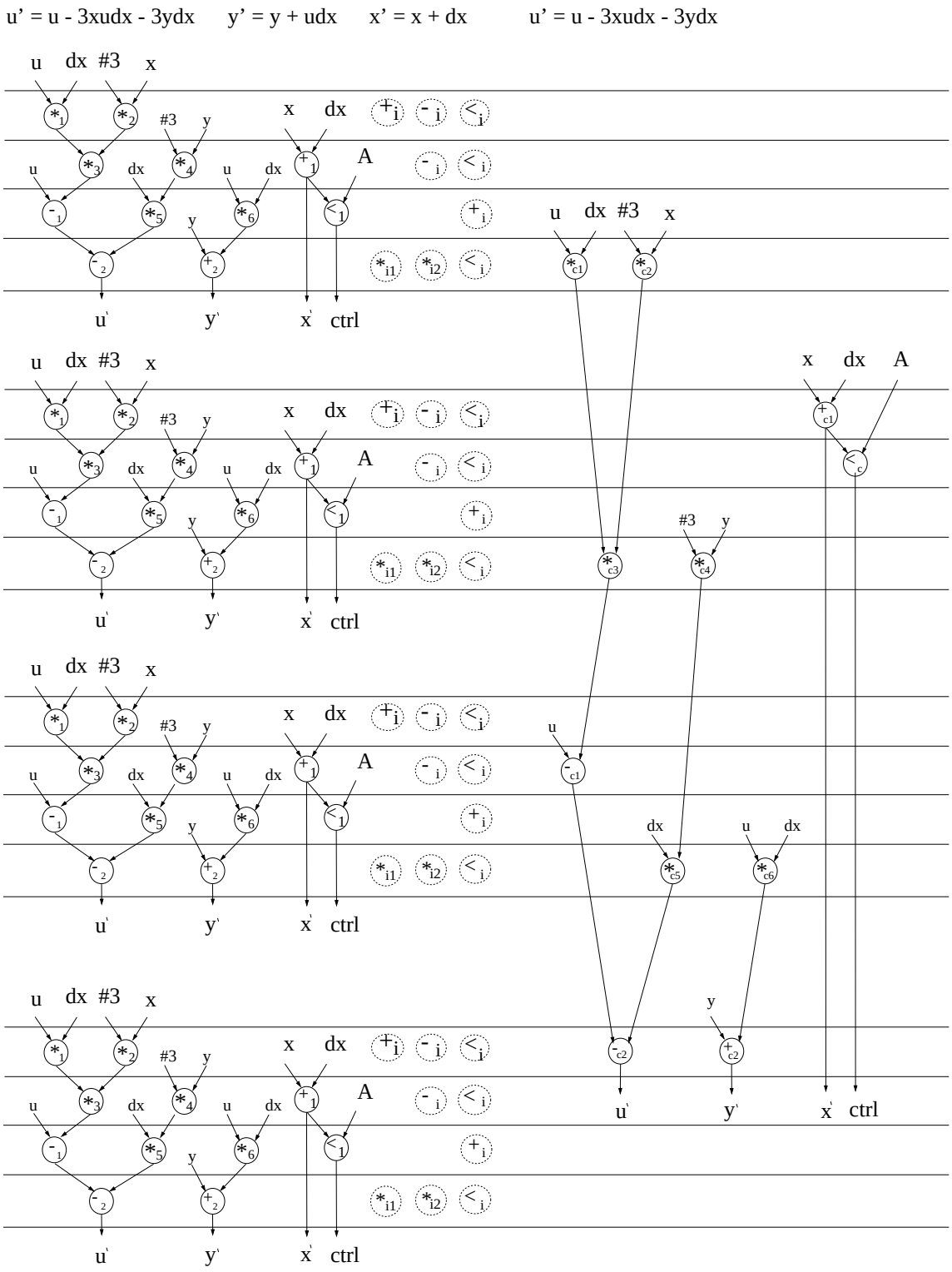


Figure 4.11: *L'architecture du test en-ligne par le graphe nominal.*

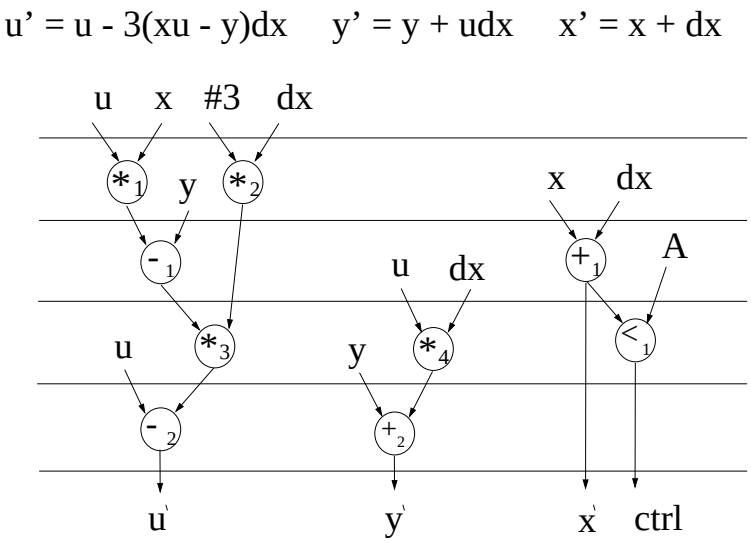


Figure 4.12: *Une variante-duale du graphe nominal de l'équation différentielle.*

Ctrl	$Mult_1$	$Mult_2$	Add	Sub	$<$	$Mult_1$	$Mult_2$	Add	Sub	$<$
1	$*_1$	$*_2$	$+_{c2}$	$-_{c2}$	$\times$	$*_1$	$*_2$	$+_{c2}$	$-_{c2}$	$\times$
2	$*_3$	$*_4$	$+_1$	$\times$	$\times$	$*_3$	$*_4$	$+_1$	$\times$	$\times$
3	$*_5$	$*_6$	$\times$	$-_1$	$<$	$*_5$	$*_6$	$\times$	$-_1$	$<$
4	$*_{c2}$	$*_{c1}$	$+_2$	$-_2$	$\times$	$*_{c2}$	$*_{c1}$	$+_2$	$-_2$	$\times$
5	$*_1$	$*_2$	$+_{c1}$	$\times$	$\times$	$*_1$	$*_2$	$+_{c1}$	$\times$	$\times$
6	$*_3$	$*_4$	$+_1$	$\times$	$<_c$	$*_3$	$*_4$	$+_1$	$-_{c1}$	$<_c$
7	$*_5$	$*_6$	$\times$	$-_1$	$<$	$*_5$	$*_6$	$\times$	$-_1$	$<$
8	$*_{c4}$	$*_{c3}$	$+_2$	$-_2$	$\times$	$*_{c4}$	$*_{c3}$	$+_2$	$-_2$	$\times$
9	$*_1$	$*_2$	$\times$	$\times$	$\times$					
10	$*_3$	$*_4$	$+_1$	$-_{c1}$	$\times$					
11	$*_5$	$*_6$	$\times$	$-_1$	$<$					
12	$*_{c6}$	$*_{c5}$	$+_2$	$-_2$	$\times$					
Avec le graphe nominal						Avec la variante-duale				

Table 4.2: *L'allocation des architectures testables en-ligne.*

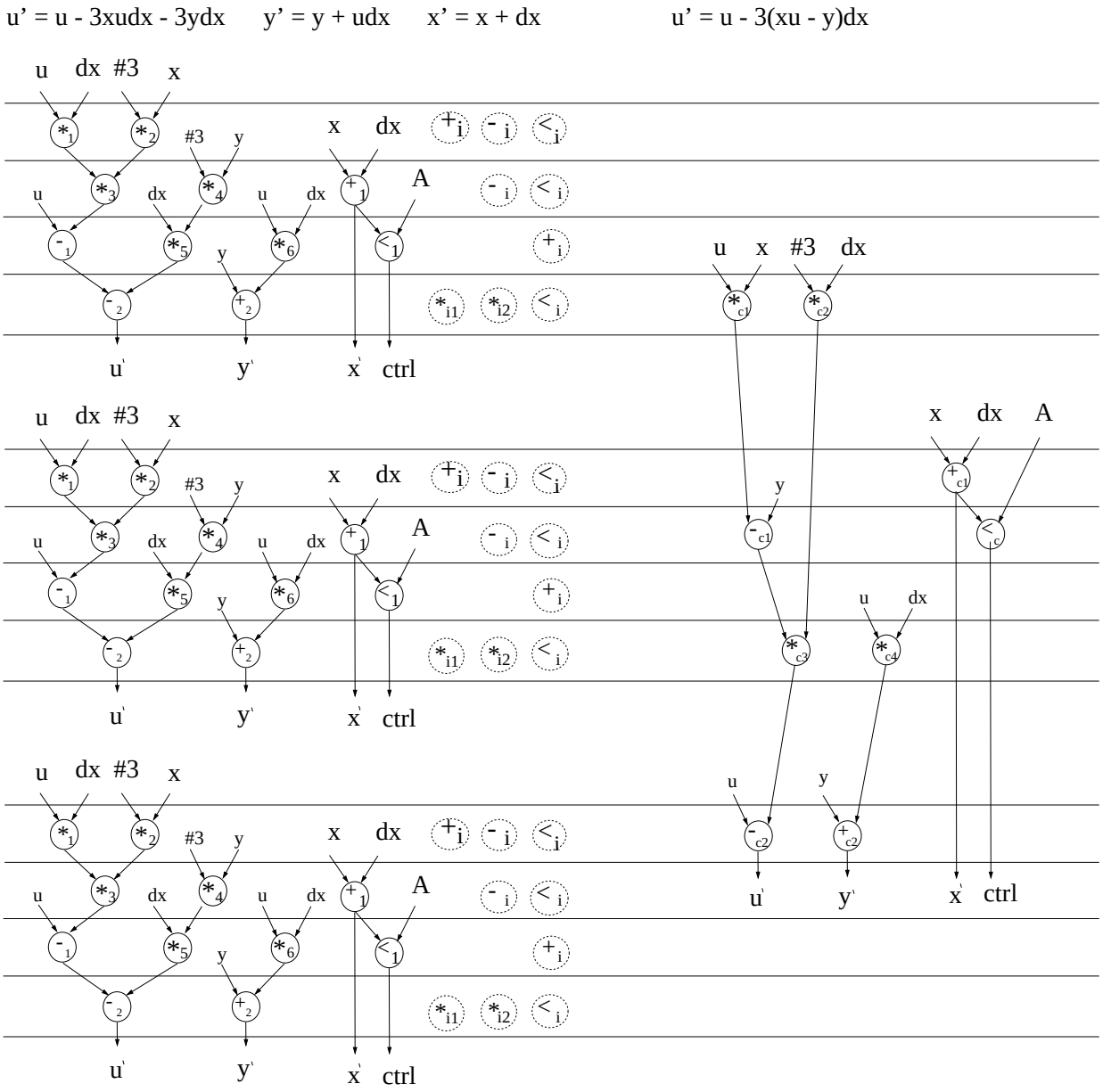


Figure 4.13: *L'architecture du test en-ligne par la variante-duale du graphe nominal.*

latence de faute du graphe concerné.

L'ordonnancement du chemin critique permet le calcul de la latence de faute obtenue par le graphe de test en-ligne. L'évaluation de la testabilité d'un graphe de test en-ligne passe par les étapes suivantes :

1. Identification des opérations du chemin critique ;
2. Ordonnancement de ces opérations jusqu'à concurrence des ressources disponibles ;
3. Calcul du nombre de cycles fonctionnels utilisés.

Ce calcul de la latence de faute est la première étape dans l'insertion du graphe de test en-ligne dans l'architecture nominale. Après l'ordonnancement des opérations du chemin critique, si la latence de faute obtenue est satisfaisante, l'ordonnancement se continue. Sinon, un autre graphe de test en-ligne est évalué. Cette évaluation de la latence de faute sera examinée en détail dans la section suivante.

4.2.5 Ordonnancement du graphe de test en-ligne

L'architecture testable en-ligne est générée par le regroupement de deux graphes de flot de donnée. Ce sont le graphe nominal et celui du test en-ligne. Le graphe de test en-ligne est ordonnancé dans les temps oisifs de l'ordonnancement nominal. A priori, les opérations de test en-ligne ne sont ordonnancées que dans les temps oisifs. Ceci réduit au minimum le surcoût en surface de la méthode. Cependant, des unités fonctionnelles supplémentaires peuvent être ajoutées si la latence de faute n'est toujours pas satisfaite. Les contraintes de surface doivent être respectées. Cela implique l'introduction partielle de la technique de redondance matérielle à côté de la technique de redondance temporelle pour la réalisation de l'architecture testable en-ligne. Deux cas sont donc à considérer lors de l'ordonnancement du graphe de test en-ligne.

Le premier est le cas où les contraintes de surface ne permettent pas l'ajout d'unités fonctionnelles supplémentaires pour le graphe de test en-ligne. Dans ce cas il faut appuyer sur la structure et l'ordonnancement du graphe de test en-ligne pour répondre aux contraintes de test en-ligne. Un ensemble de variante-duales du graphe de flot de données nominal peut être exploré pour rechercher le graphe de test en-ligne qui convient aux contraintes imposées.

Le deuxième cas est celui où les contraintes de surface permettent l'ajout de nouvelles unités fonctionnelles pour le graphe de test en-ligne. Dans ce cas, il est possible de répondre à des contraintes de test en-ligne plus critiques mais avec une dégradation de surface. Le nombre des unités fonctionnelles supplémentaires est ajouté au nombre des temps oisifs des unités fonctionnelles nominales pour formuler une nouvelle contrainte de surface.

L'ordonnancement du graphe de test en-ligne se fait en deux étapes. La première étape comprend l'ordonnancement des opérations du chemin critique. A la fin de cette étape la latence de faute peut être calculée. Si les contraintes de test en-ligne sont satisfaites, la deuxième étape prend effet. Sinon, un autre graphe de test en-ligne est sélectionné.

L'ordonnancement orienté par liste est utilisé pour insérer les opérations du graphe de test en-ligne dans l'architecture nominale. Une liste des opérations est établie. Cette liste contient les opérations du graphe de test en-ligne dans un ordre de priorité. Une opération est prioritaire si son intervalle de positionnement est critique. Un intervalle de positionnement, ou de

mobilité, d'une opération est compris entre les ordonnancements le plutôt possible (ASAP) et le plus tard possible (ALAP) du graphe nominal. La mobilité d'une opération représente le nombre de pas de contrôle auxquels cette opération peut être associée. Cette mobilité est comprise entre 1 et le nombre total de pas de contrôle de l'ordonnancement. Les opérations ayant le même degré de mobilité sont classées selon leur relation de précédence. Ce premier établissement de la liste des opérations permet l'identification du chemin critique du graphe de test en-ligne. Une fois que les opérations du chemin critique sont ordonnancées, la testabilité en-ligne est évaluée. Si la latence de faute obtenue ne correspond pas aux contraintes de test en-ligne, la décision est prise d'ajouter des unités fonctionnelles supplémentaires ou d'explorer d'autres graphes de test en-ligne. Cela dépend des contraintes de surface.

Si la latence de faute est satisfaisante, la mobilité des opérations restant à ordonnancer est donc recalculée en tenant compte du nouveau nombre de pas de contrôle et de la distribution des temps oisifs dans l'ordonnancement nominal[1].

En général, l'ordonnancement du graphe de test en-ligne se fait par les étapes suivantes:

1. Effectuer les ordonnancements ASAP et ALAP du graphe de test en-ligne ;
2. Calculer les mobilités des opérations ;
3. Classer les opérations par mobilité croissante ;
4. Ordonnancer les opérations du chemin critique ;
5. Si la latence de faute obtenue est satisfaisante, continuer ;
6. Calculer les mobilités des autres opérations selon le nouveau nombre de pas de contrôle et la distribution des temps oisifs.
7. Classer les opérations par mobilité croissante, puis par ordre de précédence ;
8. Ordonnancer les opérations en donnant la priorité aux opérations de mobilité faible jusqu'à concurrence des ressources disponibles.

Les ressources disponibles sont exprimées par les temps oisifs des unités fonctionnelles de l'architecture nominale. Si les contraintes de surface le permettent, il est possible d'ajouter un certain nombre d'unités fonctionnelles pour améliorer la latence de faute. Dans ce cas, les unités fonctionnelles supplémentaires sont ajoutées comme des temps oisifs dans tous les pas de contrôle de l'ordonnancement nominal.

Après chaque ordonnancement d'une opération dans la liste, la mobilité de ses prédécesseurs et ses successeurs non ordonnancés est recalculée et la liste des opérations à ordonnancer est modifiée. Il est possible qu'une partie seulement des temps oisifs dans chaque pas de contrôle soit utilisée. Cela dépend de la structure du graphe de test en-ligne et de la distribution des temps oisifs dans l'architecture nominale. Lorsque les temps oisifs restant dans le cycle fonctionnel ne correspondent plus à l'opération sélectionnée pour l'ordonnancement ou lorsqu'il n'existe plus de temps oisifs dans le cycle fonctionnel actuel, un nouveau cycle fonctionnel est inséré. L'algorithme dans la figure (4.14) représente l'ordonnancement du graphe de test en-ligne dans l'architecture nominale.

[1]Une opération peut être associée à un pas de contrôle si ce pas appartient à l'intervalle de positionnement de l'opération et contient les temps oisifs correspondants.

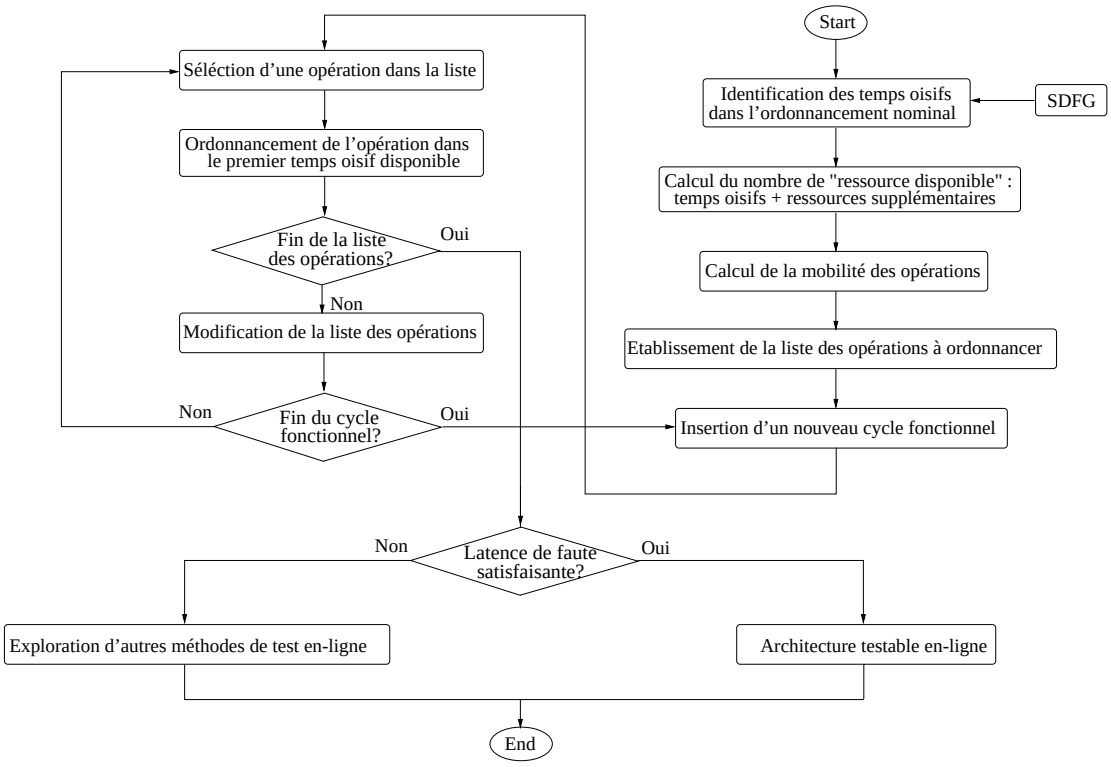


Figure 4.14: *L'ordonnancement du graphe de test en-ligne dans les temps oisifs.*

Exemple

Considérons l'exemple dans la figure (4.12). Ce graphe de test en-ligne doit être inséré dans l'architecture nominale dont l'ordonnancement est dans la figure (4.10). Il représente une variante-duale du graphe nominal. Les ordonnancements ASAP et celui ALAP du graphe de test en-ligne sont présentés dans la figure (4.15). En premier temps, les mobilités des opérations sont calculées comme le montre le tableau (4.3). Une liste des opérations à mobilité croissante est établie. L'ordre de précédence est pris en compte pour classer les opérations ayant le même degré de mobilité.

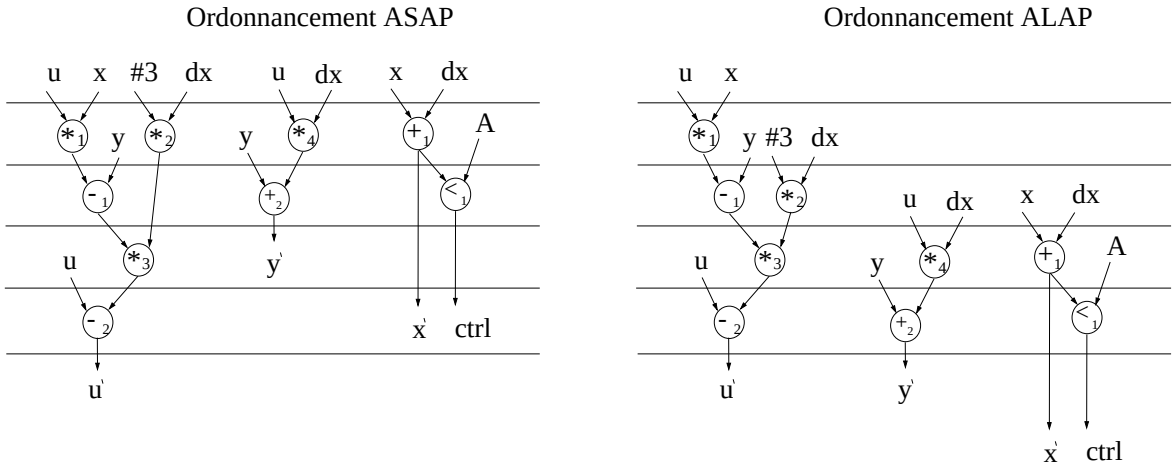


Figure 4.15: *L'ordonnancement ASAP et ALAP du graphe de test en-ligne.*

Liste des opérations	$*_1$	$-_1$	$*_3$	$-_2$	$*_2$	$*_4$	$+_2$	$+_1$	$<_1$
Mobilité	1	1	1	1	2	3	3	3	3

Table 4.3: *La mobilité des opérations du graphe de test en-ligne.*

La première opération à ordonnancer est l'opération $*_1$. Un premier cycle fonctionnel est introduit dans l'architecture testable en-ligne. Le premier pas de contrôle contenant un temps oisif qui correspond à l'opération sélectionnée se trouve en pas 4 du cycle introduit. L'opération $*_1$ est ordonnancée dans ce pas de contrôle. La mobilité ne change pas pour ses successeurs.

La deuxième opération est sélectionnée, c'est l'opération $-_1$. Celle-ci a une relation de précédence avec l'opération $*_1$, il n'existe plus de temps oisifs qui correspondent à ce type d'opérations dans le cycle fonctionnel actuel. Un nouveau cycle fonctionnel est introduit. L'opération sélectionnée peut être ordonnancée dans le premier ou le deuxième pas de contrôle. De même façon, l'opération $*_3$ est ordonnancée dans le quatrième pas de contrôle et un nouveau cycle fonctionnel est introduit pour ordonnancer l'opération $-_2$.

Au niveau de cette étape, les opérations du chemin critiques ont été ordonnancées. Le nombre de cycles fonctionnels introduits détermine la latence de faute. Supposons que la latence de faute obtenue soit satisfaisante. L'ordonnancement du graphe de test en-ligne continue. Les mobilités des opérations restant à ordonnancer sont recalculées. Le nouveau nombre de pas de contrôle et la distribution des temps oisifs dans les cycles fonctionnels introduits sont considérés. Par exemple, l'opération $*_2$ a une mobilité de 1. En effet, cette opération ne peut être ordonnancée que dans le quatrième pas de contrôle du premier cycle fonctionnel. Il existe deux raisons pour cette valeur de mobilité, la relation de précédence avec l'opération $*_3$ et l'indisponibilité de temps oisifs pour la multiplication dans les trois premiers pas de contrôle du cycle fonctionnel. La nouvelle liste des opérations avec leurs mobilités est en tableau (4.4).

Liste des opérations	$*_2$	$*_4$	$+_2$	$+_1$	$<_1$
Mobilité	1	1	1	3	3

Table 4.4: *Les nouvelles valeurs de la mobilité des opérations.*

L'opération suivante, à ordonnancer, est donc $*_2$. Cette opération a une relation de précédence avec les opérations déjà ordonnancées. L'exploration des temps oisifs qui conviennent pour l'ordonnancement de cette opération commence par le premier cycle fonctionnel. Le quatrième pas de contrôle convient car il reste encore un temps oisif pour la multiplication. L'ordonnancement de cette opération n'affecte pas l'ordre des opérations dans la liste. L'opération $*_4$ est sélectionnée pour l'ordonnancement. Elle est ordonnancée dans le quatrième pas de contrôle du deuxième cycle fonctionnel. Un successeur existe. Sa mobilité ne change pas. A noter que, après l'ordonnancement du chemin critique, les pas de contrôle associés au graphe de test en-ligne sont entre le quatrième pas de contrôle du premier cycle fonctionnel et le premier pas de contrôle du troisième cycle fonctionnel. Par conséquent, la mobilité de l'opération $+_2$ devient 1 et cette opération devient prioritaire pour l'ordonnancement. Le pas de contrôle auquel l'opération $+_2$ peut être associée contient un temps oisif pour l'addition. Dans le cas où le (ou les pas) de contrôle ne contiennent pas les temps oisifs nécessaires, les pas de contrôle suivants sont explorés et un nouveau cycle fonctionnel est introduit si nécessaire.

Les opérations $+_1$ et $<_1$ peuvent être ordonnancées dans le deuxième cycle fonctionnel. On a le choix entre les deux premiers et les deux derniers pas de contrôle.

A la fin de cette tâche d'ordonnancement, l'architecture testable en-ligne peut être générée par l'allocation de ressources et l'assignation.

L'insertion d'un graphe de test en-ligne pour la vérification globale du calcul entraîne un surcoût en surface. Cela revient au fait de l'utilisation de quelques multiplexeurs pour le chemin de données de test et à la modification nécessaire en contrôle. Ce coût reste cependant très faible en raison du coût très faible des multiplexeurs et de la patrie contrôle par rapport aux additionneurs et aux multiplieurs. A titre d'exemple, si l'on utilise 10 multiplexeurs supplémentaires pour assurer le test en-ligne d'une architecture composée de deux multiplieurs et d'un additionneur, le coût total de ces multiplexeurs ne dépasse pas

10% de la surface des multiplieurs et de l'additionneur (voir figure (3.6), de largeur 16 bit des unités fonctionnelles).

4.3 Conclusion

Dans cette partie, deux méthodes de test en-ligne ont été proposées. La méthode de test en-ligne non-concurrent et la méthode de test en-ligne semi-concurrent. L'étape de l'intégration de ces deux méthodes de test en-ligne au nouveau flot de synthèse de haut niveau pour le test en-ligne est montrée dans la figure (4.16).

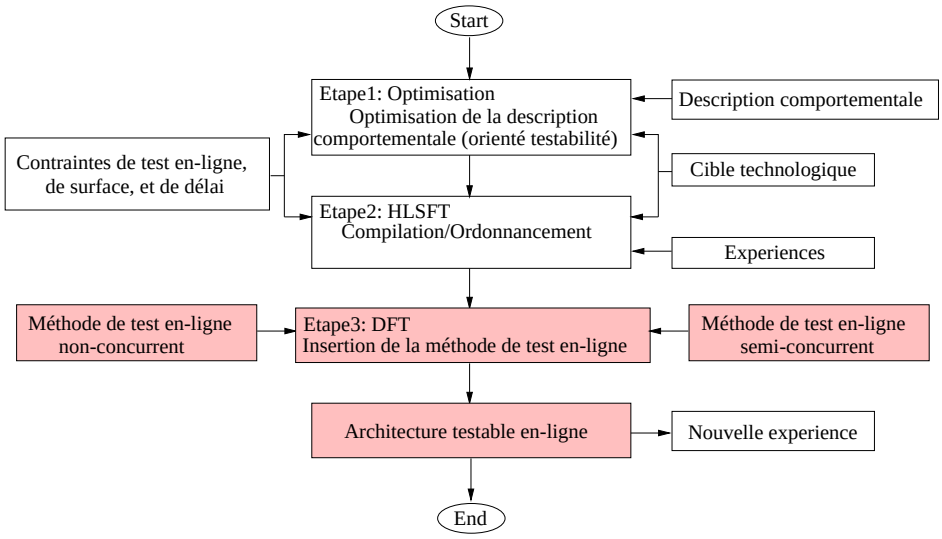


Figure 4.16: *L'intégration des méthodes de test en-ligne dans le flot de la synthèse de haut niveau pour le test en-ligne (HLS_OLT).*

Dans la partie suivante, une nouvelle approche de la synthèse de haut niveau pour le test en-ligne est présentée. Elle consiste, dans un premier temps, à effectuer une optimisation orientée testabilité en-ligne, (chapitre 5). Ensuite, une phase de compilation, (chapitre 6), permet la génération d'un graphe de flot de données ordonnancé qui tient compte des contraintes de test en-ligne, de surface et de délai.

Partie III

Nouvelle approche de HLS_OLT

Chapitre 5

Optimisation de la description comportementale pour le test en-ligne

Dans ce chapitre, une optimisation basée sur la factorisation des équations arithmétiques est présentée. Cette optimisation intervient avant les étapes de la synthèse de haut niveau et permet l'amélioration de la testabilité en-ligne du système. Elle permet également de minimiser la surface du circuit final et d'améliorer ses performances. L'amélioration de la testabilité du système est notable aussi bien pour le test en-ligne que pour le test hors-ligne. Pour le test en-ligne, la disponibilité des temps oisifs est favorisée pendant la période fonctionnelle du système afin de permettre l'application d'un ensemble de vecteurs de test qui assurent un test complet des unités fonctionnelles. Pour le test hors-ligne, l'élimination des boucles dans le circuit au niveau RTL est visée. En plus, par factorisation, d'autres solutions possibles du système au niveau ordonnancement peuvent être examinées. Cela augmente la probabilité de trouver une solution pour les contraintes imposées.

La méthode présentée dans la suite de ce chapitre considère des blocs fonctionnels constitués des opérations arithmétiques. Le problème de la factorisation est modélisé par un graphe appelé le Graphe de Dépendance de Données (DDG). Ce graphe, $DDG(V, E)$, est un graphe bipartie orienté où V représente l'ensemble des sommets et E représente l'ensemble des arêtes. Un sommet $v \in V$ correspond à une variable ou au produit de deux variables. Il existe une arête $e(v_i, v_j) \in E$ de v_i à v_j si et seulement si la variable v_j est le produit d'une opération de multiplication qui a la variable v_i comme entrée. Un graphe DDG simple est obtenu quand une seule arête sort de chaque sommet source. Un graphe DDG simple contient le minimum de nombre d'opérations nécessaires pour réaliser la fonction arithmétique.

5.1 Matrice de corrélation de variables

La matrice de corrélation de variables $VCM[N_{var}, N_m]$ guide la factorisation dans la sélection des variables communes. Le nombre de lignes N_{var} dans cette matrice correspond au nombre de variables dans la fonction à factoriser. Le nombre de colonnes N_m est le nombre de monômes constituant la fonction. A l'intersection de chaque ligne avec une colonne, le chiffre représente le nombre de répétitions de la variable dans ce monôme. L'objectif de cette matrice est de classer les variables de la fonction pour permettre la sélection des variables communes les plus avantageuses pour la factorisation, estimer les boucles potentielles dans

le système, et résoudre le problème de la compatibilité entre les variables communes. Ces concepts sont illustrés dans la figure (5.1).

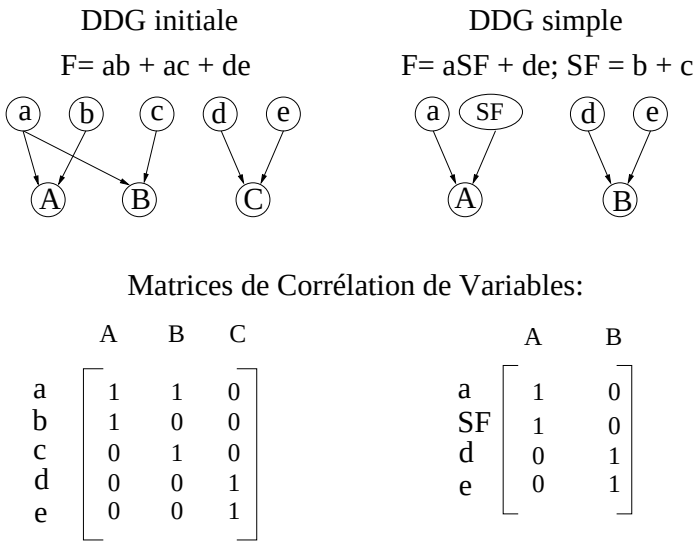


Figure 5.1: *Le concept de DDG.*

5.2 Classification des variables

Degré de la variable commune

Une variable commune peut être composée d'une ou plusieurs des variables de la fonction factorisée. Le degré de la variable commune est le nombre de variables initiales qui la composent. Par exemple, une variable commune composée de trois variables initiales est de degré trois.

Compatibilité

Une variable commune V_1 est compatible avec une autre variable commune V_2 si la sélection de V_1 pour réaliser la factorisation d'une fonction n'empêche pas la sélection de V_2 comme une autre variable commune dans la même fonction ou dans une de ses sous-fonctions pour l'itération suivante

Gain

Pour les variables communes non-compatibles, le calcul du gain apporté par l'utilisation d'une variable pour réaliser la factorisation aide à classer ces variables. Ce gain exprime les avantages apportés au circuit final en testabilité (en_ligne et hors_ligne), délai et surface. Ce calcul se fait par la formule suivante :

$$Gain(V) = D \times (M - 1)$$

où V est la variable commune, D le degré de cette variable, et M le nombre de monômes contenant V.

5.3 Boucles potentielles

En général, la factorisation minimise le nombre d'opérations à réaliser. Ceci est exploité pour améliorer la testabilité du système considéré par l'augmentation des temps oisifs dans le graphe de flot de données ordonnancé. Cependant, une factorisation réalisée dans une fonction peut, potentiellement, augmenter le nombre de boucles dans le système au niveau transfert de registre (RTL). Pour éviter ce problème, une analyse du nombre de boucles dans la fonction factorisée est effectuée. Cette analyse utilise la matrice de corrélation de variables VCM[ ].

Au niveau transfert de registre (RTL), une boucle existe si la sortie d'un opérateur est reliée directement à une de ses entrées. Au niveau ordonnancement, une boucle existe potentiellement si la sortie d'une opération est réinjectée dans la même opération[1]. Au niveau équations arithmétiques, l'existence potentielle de boucles dans le système peut être identifiée par l'analyse de la matrice de corrélation de variable VCM[ ]. En considérant l'addition et la multiplication et en supposant la disponibilité d'un additionneur et d'un multiplieur seulement, une possible formation de boucles existe si plus de deux opérations du même types se succèdent. Autrement dit, une boucle de multiplication peut exister dans un monôme m_i si la somme des éléments $VCM[0 \rightarrow N_{var}, i]$ est supérieure à 2. De même, une boucle d'addition peut exister dans une fonction ou sous-fonction si celle-ci est composée de plus de deux monômes. En général, selon les contraintes de surfaces imposées au circuit final, une estimation du nombre d'opérateurs au niveau RTL est faite pour guider l'analyse de l'existence de boucles potentielles. Par exemple, si les contraintes de surfaces permettent l'utilisation de trois multiplieurs, une boucle de multiplication peut exister si la somme des éléments de la colonne correspondante dans VCM[ ] dépasse trois. Cette estimation se fait en liaison avec la cible technologique où une base de données contenant la surface des différents opérateurs est disponible. Les exemples suivants illustrent les concepts de cette méthode.

5.4 Exemples

Considérons la fonction :

$$F(a, b, c, d, e) = abc + abd + de$$

Le graphe de dépendance de données et la matrice de corrélation de variables sont présentés dans la figure (5.2).

Cette fonction contient deux variables communes ab et d. Ces deux variables ne sont pas compatibles car la sélection de ab pour réaliser la factorisation empêche la sélection de d et vice versa. Pour résoudre ce problème de compatibilité, le gain de chaque variable est calculé. Les variables communes ab et d, avec leurs degrés et gains correspondant, sont classés dans le tableau (5.1). Les avantages de l'utilisation de la variable commune ab dans la réalisation de la factorisation seront discutés en détail dans la section 5.7.

Sous-fonction

Pour montrer la manipulation des sous-fonction par la méthode proposée, considérons la fonction suivante :

$$F(a, b, c, d, e) = abc + bce + cdf$$

[1] En fait, c'est un problème d'allocation d'unités fonctionnelles et d'assignation.

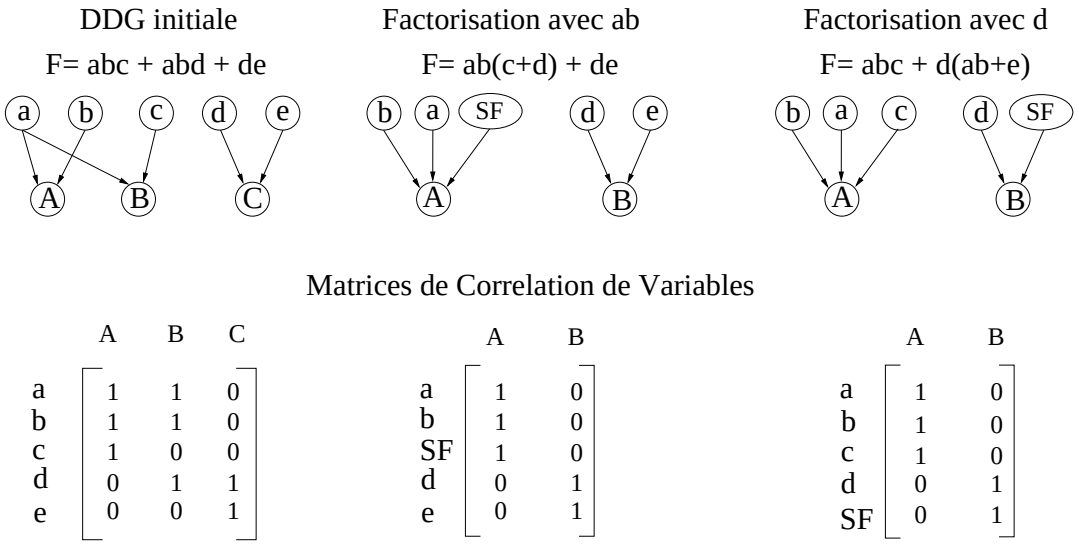


Figure 5.2: *Le graphe de dépendance de données et la matrice de corrélation de variables.*

V	D	M	$Gain(V) = D \times (M-1)$
ab	2	$abc + abd$	$2 \times (2-1) = 2$
d	2	$abd + de$	$1 \times (2-1) = 1$

Table 5.1: *Classification des variables communes.*

La sélection de la variable c comme une variable commune pour la première itération de l'algorithme n'empêchera pas la sélection de la variable b pour la sous-fonction résultante de la première factorisation. Le gain de la variable c est plus grand que celui de la variable b. Pour cela, la variable c est sélectionnée pour la première étape de la factorisation et ensuite b est choisie dans la sous-fonction résultante. L'application de la méthode pour cet exemple est illustrée dans la figure (5.3).

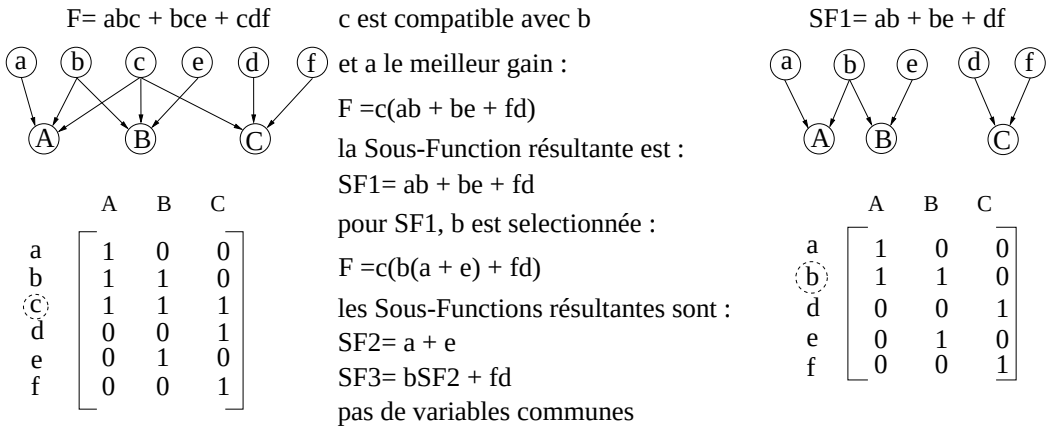


Figure 5.3: *Sous-fonctions.*

Compatibilité

La résolution du problème de compatibilité consiste à trouver la variable ou l'ensemble de variables communes qui garantissent le meilleur gain à la factorisation et qui éliminent le conflit au niveau du choix des variables communes. Pour illustrer ce concept, considérons la fonction suivante :

$$F(a, b, c, d, e, f) = abc + bce + def + ef$$

Dans cette fonction, il y a quatre variables communes possibles. Ce sont bc, e, f, et ef. Le choix des variables, qui seront utilisées pour la factorisation, est basé sur le gain des différentes variables communes. La figure (5.4) montre la comparaison des gains des différentes variables.

5.5 Algorithme de la factorisation

L'algorithme présenté dans la figure (5.5) sélectionne les meilleures variables communes parmi les variables d'une fonction $F(a, b, c, \ldots)$ pour réaliser la factorisation. Dans un premier temps, les variables de la fonction considérée sont classifiées selon leur présence dans les monômes. Ensuite, les variables, de tous les degrés, les plus présentes dans l'ensemble des monômes sont sélectionnées comme étant potentiellement les meilleures variables communes de la fonction considérée (PBCV : Potential Best Common Variables). Enfin, la compatibilité entre les variables sélectionnées est analysée. Si toutes les variables de l'ensemble PBCV sont compatibles, elles sont considérées comme les meilleures variables communes (BCV : Best

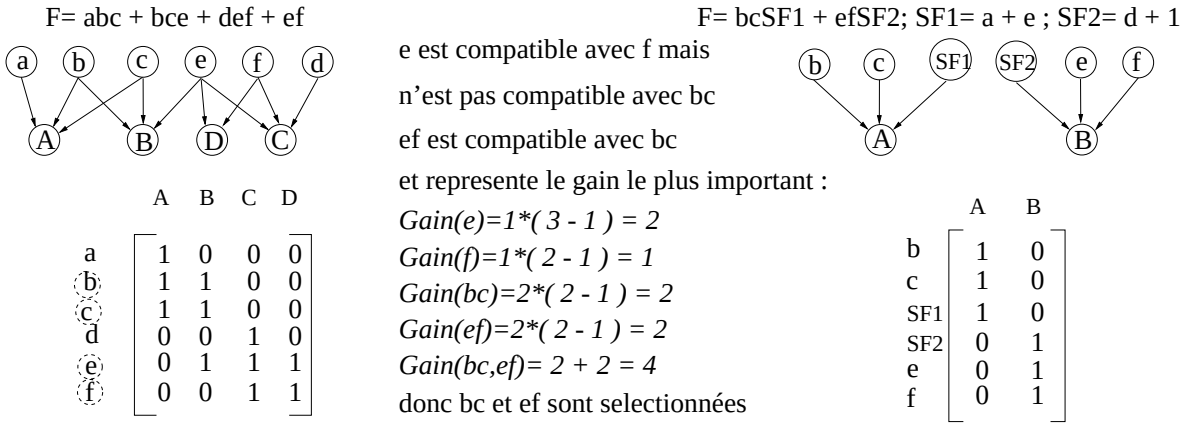


Figure 5.4: *Compatibilité.*

Common Variables) et la factorisation est réalisée. Dans le cas où un problème de variables non-compatibles existe, l'algorithme analyse le gain des différentes variables dans PBCV et sélectionne la combinaison ayant le meilleur gain comme les meilleures variables communes (BCV). Une nouvelle itération de l'algorithme est effectuée pour chacune des sous-fonctions résultantes de la factorisation précédente. Cette itération s'arête quand il n'existe plus de variables communes dans aucune sous-fonction.

Selon le résultat de l'analyse de la matrice VCM[ ], si le nombre de boucles est potentiellement plus grand dans la fonction ou la sous-fonction factorisée par rapport à celui de la fonction ou la sous-fonction initiale, la factorisation est annulée et la fonction ou la sous-fonction initiale est considérée pour le reste des étapes de notre méthode. Il est clair que le nombre exact de boucles dans le système ne peut être connu qu'au niveau transfert de registre (RTL). Par ailleurs, quelques boucles potentielles dans le système peuvent être évitées au niveau allocation des unités fonctionnelles et assignation. Néanmoins, l'objectif de la factorisation est de minimiser le nombre de boucles potentielles dans le système pour faciliter ainsi la tâche de la synthèse de haut niveau. Autrement dit, si par simple factorisation toutes les boucles sont éliminées et les contraintes de test en-ligne sont satisfaites, les efforts dans les étapes suivantes sont orientés pour satisfaire les contraintes de temps et de surface ou pour améliorer les performances.

Les exemples dans les sections suivantes illustrent l'application et les avantages de la méthode.

5.6 Amélioration de la testabilité en-ligne

Le gain apporté à la testabilité en-ligne est obtenu par l'augmentation des temps oisifs dans la période fonctionnelle. L'exemple de l'équation différentielle cité en [51] est considéré dans cette section. Les relations suivantes représentent l'équation différentielle :

$$u` = u - 3xudx - 3ydx$$
$$y` = y + udx$$
$$x` = x + dx$$

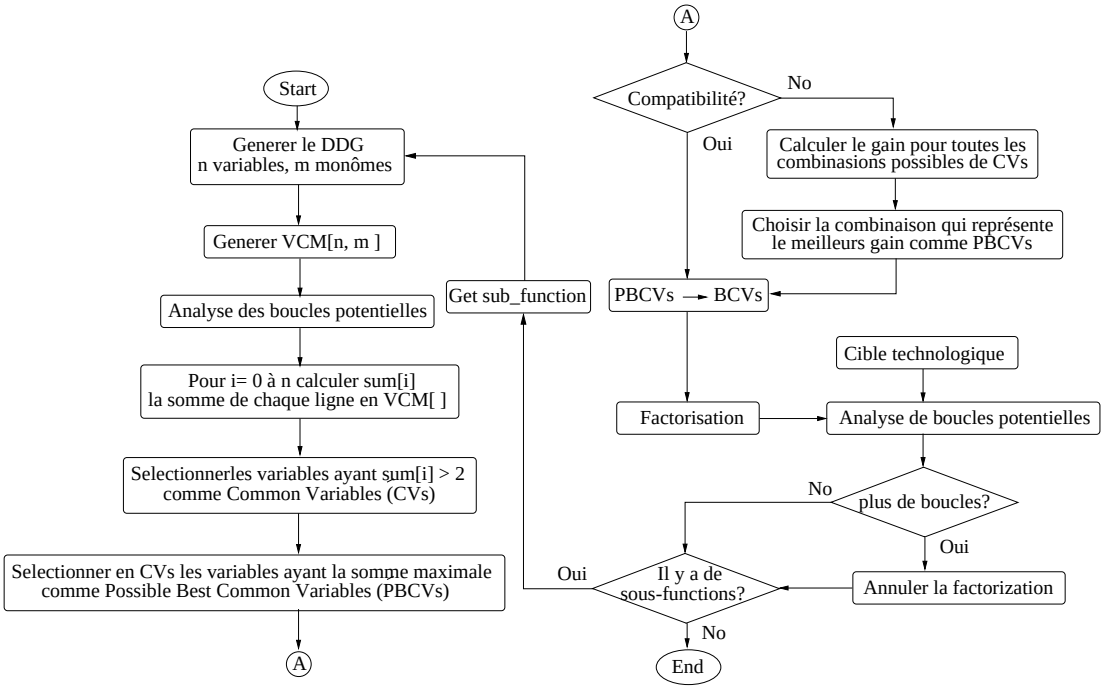


Figure 5.5: *L'algorithme de la factorisation.*

L'équation qui est considérée par la factorisation est la première équation. Dans un premier temps, la matrice de corrélation de variables VCM[ ] est construite(figure (5.6)).

Il existe deux variables communes. Ce sont u et $3dx$. Comme ces deux variables ne sont pas compatibles, le gain est calculé pour chacune d'entre eux. Cela donne :

$$Gain(3dx) = 2 \times (2 - 1) = 2$$

$$Gain(u) = 1 \times (2 - 1) = 1$$

La variable commune $3dx$ du deuxième degré est sélectionné pour la factorisation car il apporte le meilleur gain. L'équation factorisée est donc de la forme :

$$u^{'} = u - 3(xu - y)dx$$

et le graphe de dépendance de données qui lui correspond est un graphe DDG simple, (figure (5.7)).

L' amélioration apportée au système au niveau testabilité en-ligne se traduit par l'augmentation des temps oisifs pour la multiplication. Cela permet, au niveau RTL, de tester les multiplieurs deux fois dans la période fonctionnelle. De plus, une boucle potentielle dans le système a été éliminée. Dans la figure (5.8), le système est présenté avant et après l'application de l'optimisation. Les opérations de test réalisées dans les temps oisifs sont marquées par des pointillés.

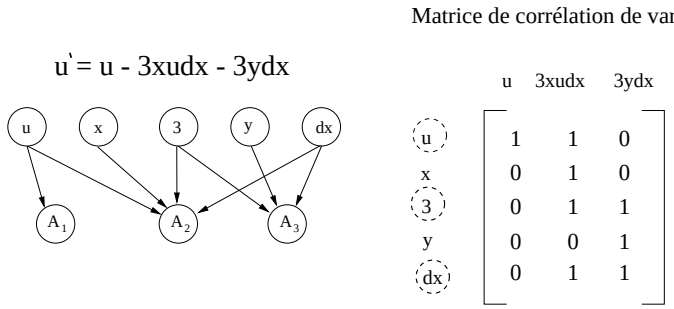


Figure 5.6: *Le graphe DDG et la matrice VCM[ ] de l'équation différentielle avant l'optimisation.*

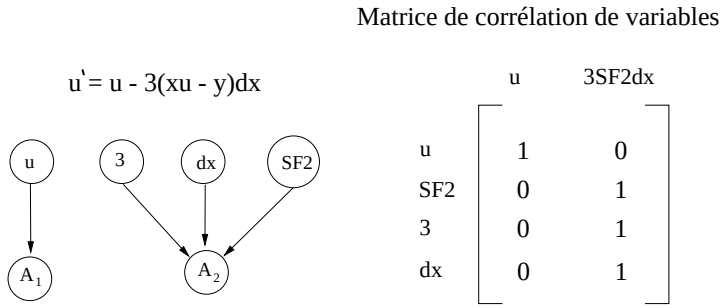


Figure 5.7: *Le graphe DDG et la matrice VCM[ ] de l'équation différentielle après l'optimisation.*

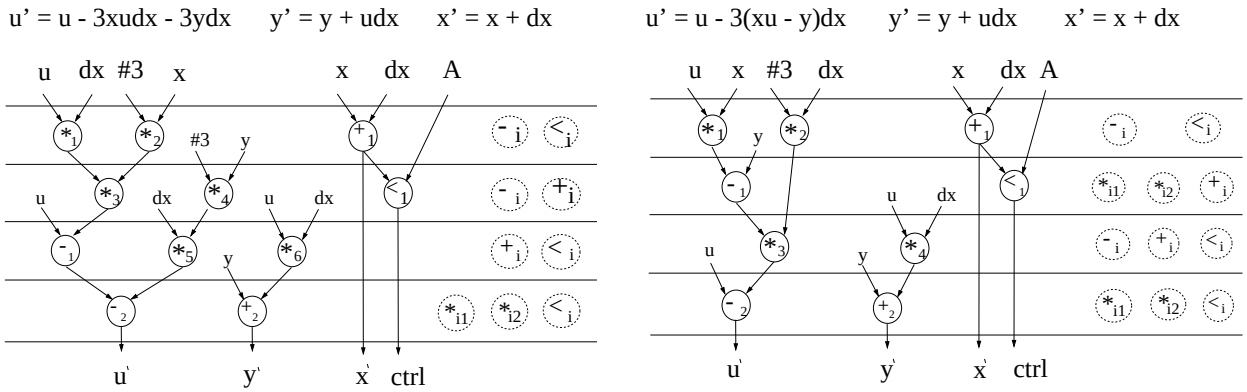


Figure 5.8: *L'amélioration de la testabilité.*

Pour avoir une idée de l'effet du choix basé sur le gain des variables communes, une comparaison entre le choix de u et le choix de $3dx$, pour réaliser la factorisation, est présenté dans la figure (5.9). On constate clairement que le choix de la variable u affecte le délai du système de $+25\%$ sans apporter un gain considérable en testabilité en-ligne. En revanche, avec le même nombre de pas de contrôle, le choix de la variable $3dx$ apporte un gain considérable en testabilité en-ligne.

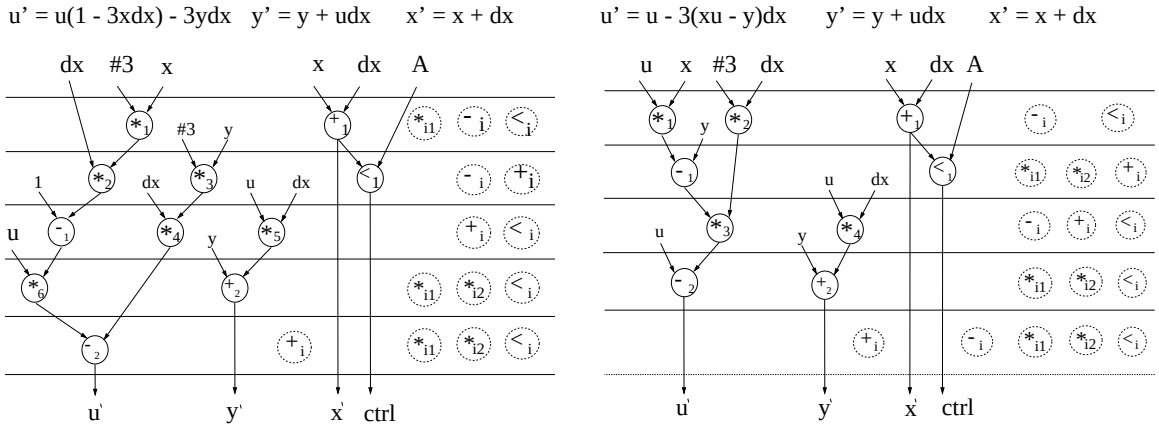


Figure 5.9: *L'effet du choix basé sur le gain.*

5.7 Enrichissement de l'espace de solutions

Un des avantages de l'optimisation proposée est l'enrichissement de l'espace de solutions au niveau DFG. Il consiste à mettre à la disposition de la deuxième étape de la méthode proposée(figure 2.2) une variété de graphes de dépendance de données pour l'ordonnancement. Cet avantage sera examiné pour la fonction de l'exemple cité dans la figure (5.2). L'ordonnancement est réalisé pour la fonction avant et après la factorisation avec les différentes variables communes.

Dans un premier temps, considérons le choix de la variable commune ab pour réaliser la factorisation. La figure (5.10) représente la fonction $f(a, b, c, d, e)$ avant et après la factorisation avec deux ordonnancements possibles pour la fonction factorisée.

Dans la figure (5.10-a) l'ordonnancement de la fonction initiale est présenté. On constate qu'il y a deux temps oisifs pour l'addition et quatre temps oisifs pour la multiplication. Etant donné le nombre minimal d'unités fonctionnelles imposé par l'ordonnancement, l'additionneur et les deux multiplieurs utilisés dans le circuit peuvent être testé deux fois chacun. Par ailleurs, Il est clair qu'une boucle sera présente au niveau RTL à cause des deux opérations $+_1$ et $+_2$ qui se succèdent. En appliquant l'optimisation proposée, les ordonnancements dans la figure (5.10 b et c) peuvent être obtenus. Cette optimisation apporte deux solutions supplémentaires. La première solution (figure 5.10-b) économise un pas de contrôle pour le même nombre d'unités fonctionnelles. La deuxième solution (figure 5.10-c) économise un multiplieur pour le même nombre de pas de contrôle. Si les contraintes de test en-ligne sont critiques et qu'il manque encore des temps oisifs, la première solution est

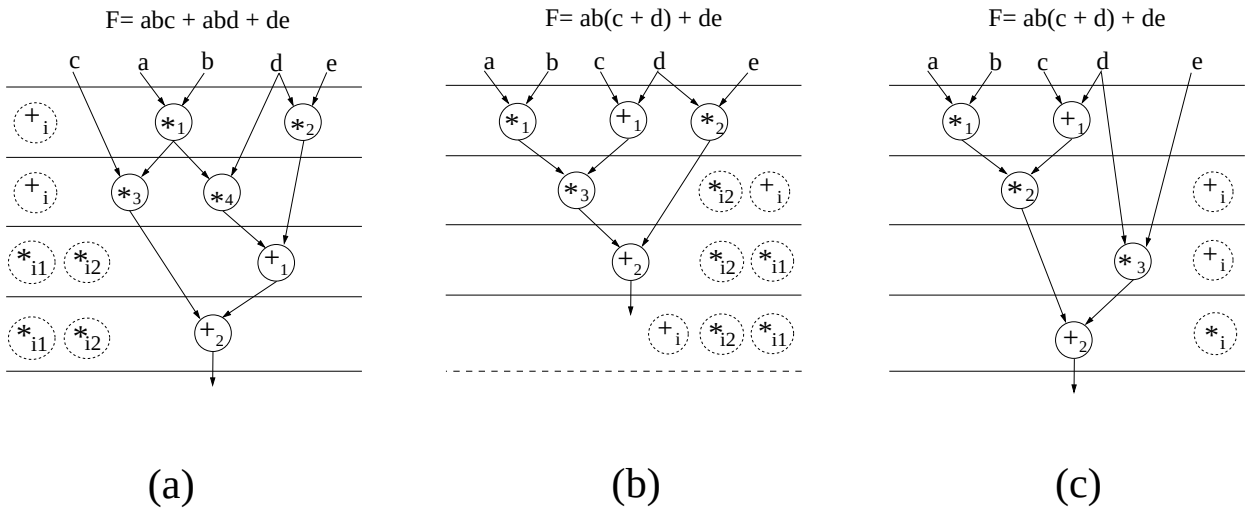


Figure 5.10: *Enrichissement de l'espace de solutions.*

utilisée tout en dévouant le dernier pas de contrôle au test en-ligne. Si les contraintes de test en-ligne sont satisfaites alors que les contraintes de délai ne le sont pas, la première solution est aussi utilisée en supprimant le dernier pas de contrôle. Si les contraintes de surface ne sont pas satisfaites, la deuxième solution est utilisée.

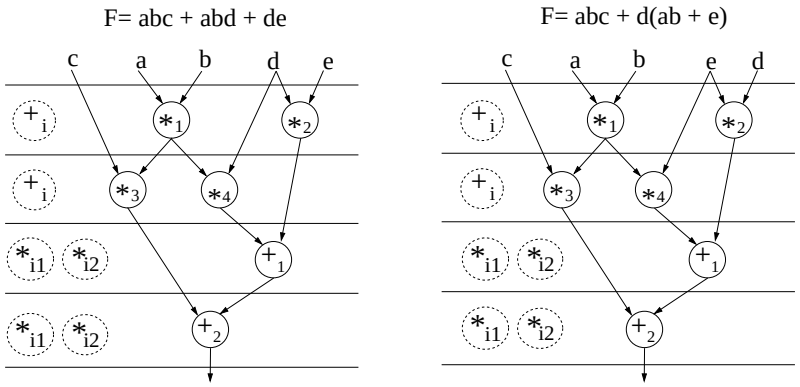


Figure 5.11: *L'effet du gain sur le choix des variables communes.*

Considérons maintenant le choix de la variable d pour la factorisation. L'ordonnancement de la fonction initiale et de celle factorisée sont présentés dans la figure (5.11). Il est clair que le choix de la variable d pour la factorisation n'apporte aucun gain ni en testabilité en-ligne, ni en surface, ni en délai. Cela montre l'importance du choix basé sur le gain des variables communes.

5.8 Elimination des boucles potentielles

Pour montrer l'efficacité de l'optimisation proposée dans l'élimination des boucles potentielles, l'exemple $ex3$, initialement évoqué par Lee *et al.* dans [65] et adopté par Singh et Knight dans [103], est utilisé. Cet exemple est une portion de la transformation discrète de huit-point du cosinus. La figure (5.12) représente la partie de la fonction avant (figure (5.12-a)) et après (figure (5.12-b)) la factorisation.

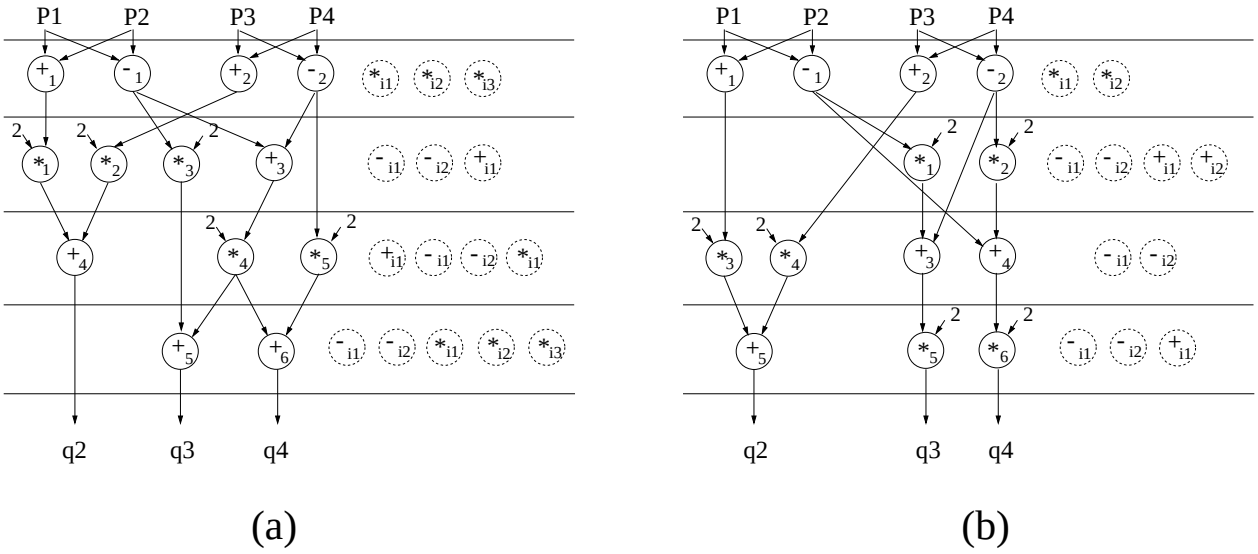


Figure 5.12: *Elimination des boucles potentielles.*

Les équations qui représentent q_2, q_3, q_4 dans la fonction initiale sont :

$q_2 = \quad 2(p_1 + p_2) + 2(p_3 + p_4)$

$q_3 = \quad 2(p_1 - p_2) + 2[(p_1 - p_2) + (p_3 - p_4)]$

$q_4 = \quad 2(p_3 - p_4) + 2[(p_1 - p_2) + (p_3 - p_4)]$

Les graphes de dépendance de données (DDGs) et les matrices de corrélation de variables (VCMs) qui correspondent à ces équations sont présentées dans la figure (5.13).

La seule variable commune dans q_2 est 2. La factorisation de q_2 donne :

$$q_2 = 2(p_1 + p_2 + p_3 + p_4)$$

La sous-fonction $SF = (p_1 + p_2 + p_3 + p_4)$ résultante dans la forme factorisée contient une série des opérations d'addition. Pour pouvoir décider si cette séquence d'opérations contient des boucles potentielles, il faut disposer d'informations sur les contraintes de surface et sur la cible technologique. Dans le cas où un seul additionneur est utilisé dans la structure RTL, cette séquence d'opérations d'addition engendrera certainement une boucle au niveau RTL. Supposons que le nombre d'additionneurs correspond au nombre maximal des opérations dans le graphe ordonnancé dans la figure (5.12-b) qui est 2. Dans ce cas, la somme des monômes dans la sous-fonction considérée est égale à 4 et une boucle existe potentiellement au niveau RTL pour l'addition. Cette analyse conduit à renoncer à la factorisation de cette

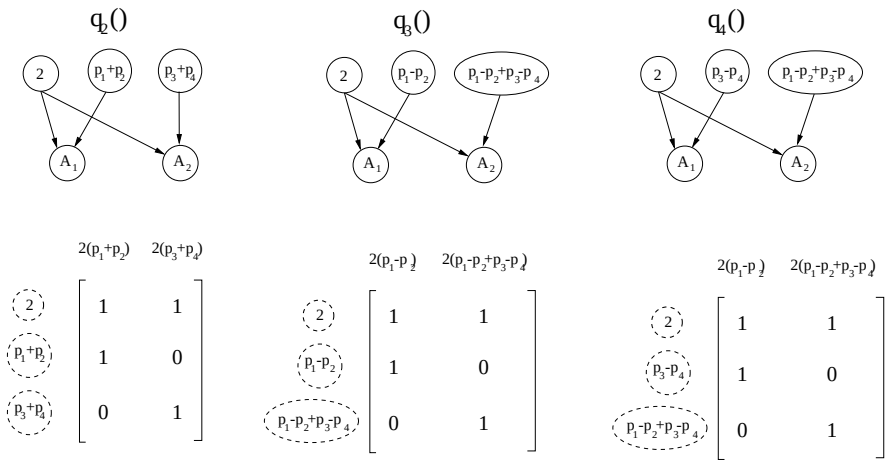


Figure 5.13: *Les DDGs et les VCMs de l'exemple ex3.*

fonction pour éviter ces boucles potentielles. La factorisation est donc réalisée seulement pour les fonctions q_3 et q_4 et ses sous-fonctions.

5.9 Application aux méthodes de test en-ligne

L'optimisation présentée dans ce chapitre est avantageuse pour plusieurs méthodes de test en-ligne utilisant la redondance temporelle dans le système. En effet, la favorisation de l'existence des temps oisifs au niveau ordonnancement améliore la testabilité en-ligne par l'amélioration de la latence de faute. Dans la suite de cette section, nous allons examiner les avantages de la factorisation pour les différentes méthodes de test en-ligne retenues dans cette étude.

5.9.1 Test en-ligne non-concurrent

Les méthodes de test en-ligne adoptées dans cette étude sont toutes basées sur l'exploitation des périodes oisives dans le système sous test. Pour la méthode de test en-ligne non-concurrent prsente dans le chapitre 3, les vecteurs de test sont appliqués dans les temps oisifs des unités fonctionnelles. Il en résulte que l'augmentation des temps oisifs réduit la latence de faute.

Si l'on considère le cas des systèmes qui disposent d'une période de repos, T_i, importante mais pas suffisante pour atteindre une latence de faute imposée par les contraintes de test en-ligne, les temps oisifs dans la période fonctionnelle, T_f, peuvent être utilisés pour compléter le nombre de vecteurs de test à appliquer. L'amélioration du test en-ligne de ce type de systèmes se fait par :

- l'augmentation des temps oisifs dans la période fonctionnelle, T_f, du système.
- l'amélioration de la latence du système par la réduction de son délai. Ceci implique l'augmentation de la période de repos T_i et ainsi l'amélioration de la latence de faute.

Les exemples dans la figures(5.8 ... 5.10) illustrent l'utilité de cette optimisation pour le test en-ligne non-concurrent.

5.9.2 Test en-ligne semi-concurrent

L'effet de la factorisation sur la méthode de test en-ligne semi-concurrent sera illustré pour l'exploitation de la distributivité et la réalisation du calcul global mais aussi pour l'exploitation des variantes-duales du graphe nominal. Pour cette ullistration, l'exemple de l'équation différentielle sera utilisé. Cette équation a été utilisée dans ce chapitre pour illustrer la factorisation et le calcul du gain des variables communes dans la section(5.6).

Exploitation de la distributivité

Il convient de rappeler que, pour cette méthode de test en-ligne semi-concurrent, il est nécessaire de disposer des entrées fonctionnelles qui vont servir à produire l'excitation pour le test en-ligne et des temps oisifs nécessaires pour l'application de cette excitation.

Considérons l'exemple de l'équation différentielle, dans la figure (5.8). La forme factorisée de la première équation permet la redistribution des temps oisifs de façon à disposer des entrées fonctionnelles pour toutes les unités fonctionnelles au deuxième pas de contrôle. La répartition des temps oisifs des unités fonctionnelles assure la disponibilité de toutes ces unités pour le test en-ligne dans les trois premiers pas de contrôle. Par conséquent, toutes les unités fonctionnelles peuvent être testées en-ligne à la fin du troisième pas de contrôle.

Contrairement à ça, la forme originale de cette équation présente deux difficultés pour ce type de test en-ligne. La première difficulté se manifeste dans le fait que les multiplieurs ne sont disponibles qu'au dernier pas de contrôle. La deuxième difficulté est dans la disponibilité des entrées fonctionnelles pour le soustracteur, qui n'apparaît qu'au troisième pas de contrôle, alors qu'aucune période oisive n'est disponible pour cet opérateur au quatrième pas de contrôle.

En outre, si nous considérons chaque unité fonctionnelle seule, toutes les unités fonctionnelles peuvent être testées au moins deux fois dans la période fonctionnelle de la forme factorisée, alors que, pour la forme non-factorisée, les multiplieurs ne seront testés qu'une seule fois dans la même période. Ces avantages de la forme factorisée sont illustrés dans la figure (5.14). Les architectures testables en-ligne de la forme factorisée et de la forme non-factorisée sont générées.

Vérification globale du calcul

La vérification du calcul global consiste à réordonnancer le graphe nominal de flot de données dans les temps oisifs de l'ordonnancement nominal. Ceci sert à faire le même calcul deux fois pour comparer les résultats et assurer le bon fonctionnement du système. Ce nouvel ordonnancement concerne donc le graphe du test en-ligne. L'architecture finale composée du graphe nominal et de celui du test en-ligne est une architecture testable en-ligne. Considérons encore l'exemple de l'équation différentielle. La réalisation de l'architecture testable en-ligne avec le système des équations avant la factorisation a donné le résultat dans la figure (4.11) avec $L_F \geq 12$ pas de contrôle. La forme factorisée la plus favorable pour la testabilité en-ligne est celle qui utilise la variable commune $3dx$ car elle apporte le meilleur gain. En utilisant cette forme pour générer l'architecture nominale et celle du test en-ligne, la latence de faute est réduite à $L_F \geq 4$ pas de contrôle. Un gain de 66% en latence de faute est donc obtenu.

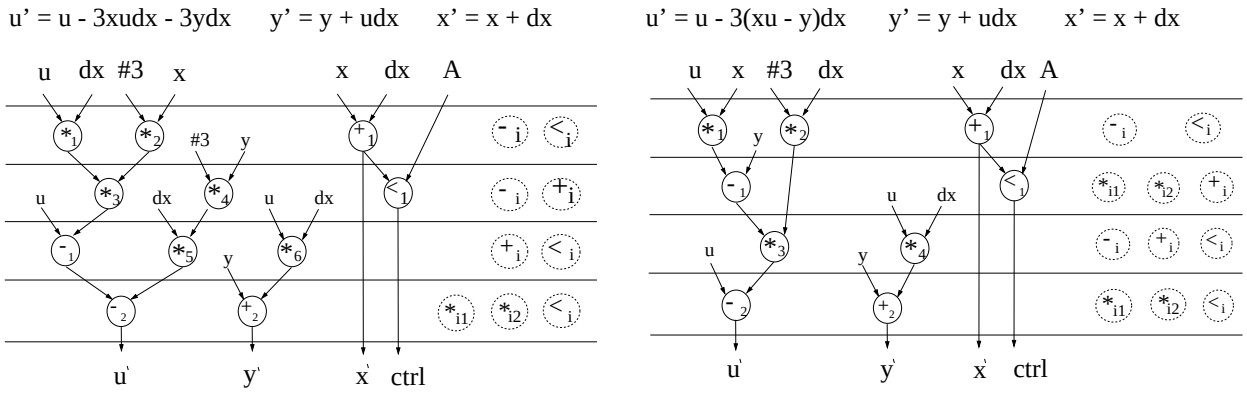


Figure 5.14: *Les architectures testables en-ligne : avantages de la factorisation.*

L'ordonnancement du graphe nominal et du graphe du test en-ligne de la forme factorisée est dans la figure (5.15). L'allocation des unités fonctionnelles pour l'architecture testable en-ligne est dans le tableau (5.2).

Ctrl	$Mult_1$	$Mult_2$	Add	Sub	$<$
1	$*_1$	$*_2$	$+_c$	$-_c$	$\times$
2	$*_{c2}$	$*_{c1}$	$+$	$-$	$\times$
3	$*_1$	$*_2$	$+_c$	$-_c$	$<$
4	$*_{c2}$	$*_{c1}$	$+$	$-$	$<_c$

Table 5.2: *L'allocation des unités fonctionnelles pour l'architecture testable en-ligne.*

Exploitation des variantes-duales du graphe de flot de données

L'utilité de l'exploitation d'une variante-duale de graphe de flot de données pour le test en-ligne a été explorée dans le chapitre 4. Le même exemple que celui utilisé pour illustrer l'implémentation de la méthode dans ce chapitre-là sera utilisé dans cette section. L'objectif ici est de montrer l'avantage de l'exploitation d'une variante-duale du graphe de flot de données nominal pour le test en-ligne. L'exemple qui a été utilisé dans la section 4.2.3 est l'équation différentielle. Une forme factorisée du graphe nominal a été ordonnancée dans les temps oisifs de façon à ne pas affecter le coût en surface du circuit final. Si les contraintes de surface permettent l'ajout d'un multiplieur, la latence de faute peut être encore améliorée de façon significative. La variante-duale du graphe de flot de données de la première équation représente, aussi dans ce cas, une importance. La réalisation de l'architecture de test en-ligne par le graphe nominal donne une latence de faute $L_F \geq 8$ pas de contrôle, alors que cette architecture réalisée par le graphe de flot de données de l'équation factorisée donne

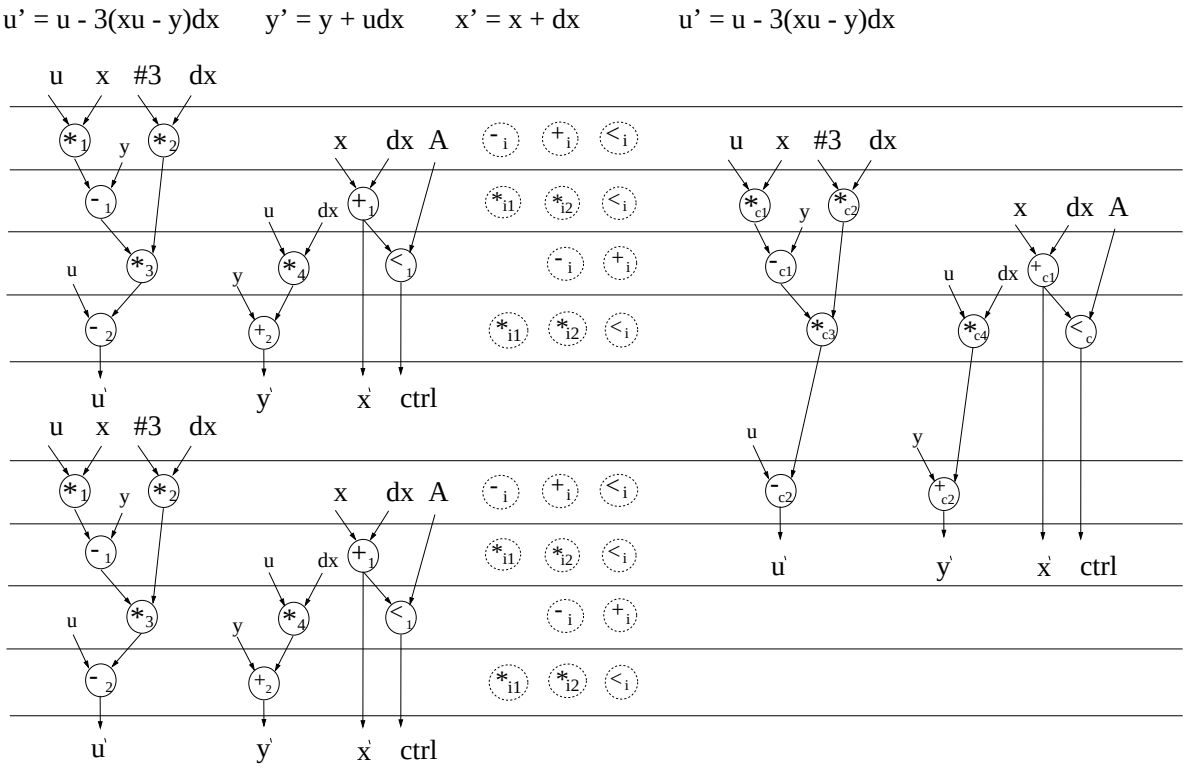


Figure 5.15: *L'effet de la factorisation sur la méthode de la vérification globale du calcul.*

une latence de faute $L_F \geq 5$ pas de contrôle[2], ce qui fait une amélioration de la testabilité en-ligne de 37.5%. La figure (5.16) illustre cet avantage. A noter que l'association de la redondance matérielle à la redondance temporelle dans ce type de test en-ligne réduit la possibilité de masquer les fautes. Ceci apporte une amélioration significative à la qualité du test en-ligne. L'allocation des unités fonctionnelles pour les architectures testables en-ligne est dans le tableau (5.3).

Ctrl	$Mult_1$	$Mult_2$	$Mult_3$	Add	Sub	$<$	$Mult_1$	$Mult_2$	$Mult_3$	Add	Sub	$<$
1	$*_1$	$*_2$	$*_{c1}$	$+_{c1}$	$\times$	$\times$	$*_1$	$*_2$	$*_{c1}$	$\times$	$\times$	$\times$
2	$*_3$	$*_4$	$*_{c2}$	$+_1$	$\times$	$<_c$	$*_3$	$*_4$	$*_{c2}$	$+_1$	$-_{c1}$	$\times$
3	$*_5$	$*_6$	$*_{c3}$	$\times$	$-_1$	$<$	$*_5$	$*_6$	$*_{c3}$	$+_{c1}$	$-_1$	$<$
4	$\times$	$*_{c4}$	$*_{c5}$	$+_2$	$-_2$	$\times$	$\times$	$\times$	$*_{c4}$	$+_2$	$-_2$	$\times$
5	$*_1$	$*_2$	$*_{c6}$	$+_{c2}$	$-_{c1}$	$\times$	$*_1$	$*_2$	$*_{c1}$	$+_{c2}$	$-_{c2}$	$\times$
6	$*_3$	$*_4$	$\times$	$+_1$	$-_{c2}$	$\times$						
7	$*_5$	$*_6$	$\times$	$\times$	$-_1$	$<$						
8	$\times$	$\times$	$\times$	$+_2$	$-_2$	$\times$						
la forme non-factorisée							la forme factorisée					

Table 5.3: *L'allocation des architectures testables en-ligne avec dégradation de surface.*

5.10 Conclusion

Dans ce chapitre, une optimisation orientée testabilité en-ligne a été proposée. Cette optimisation concerne les équations arithmétiques dans une description comportementale. Ces équations sont factorisées en utilisant les variables communes qui garantissent le meilleur gain en test en-ligne. L'impact de cette optimisation sur les différentes méthodes de test en-ligne non-concurrent et semi-concurrent a été analysé. L'intégration de cette optimisation dans le flot général de HLS_OLT proposé par cette étude est présentée dans la figure (5.17).

[2]Le graphe du test en-ligne recommence au premier pas de contrôle dans la deuxième itération du graphe nominal.

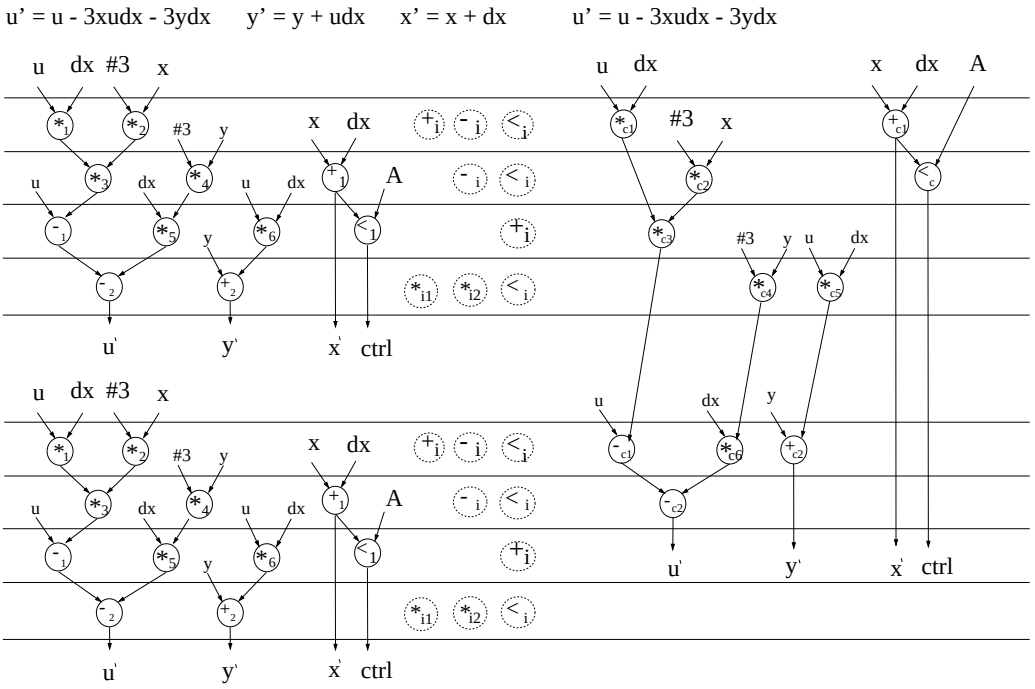


Figure 5.16: *Le graphe nominal et sa variante-duale avec dégradation de surface.*

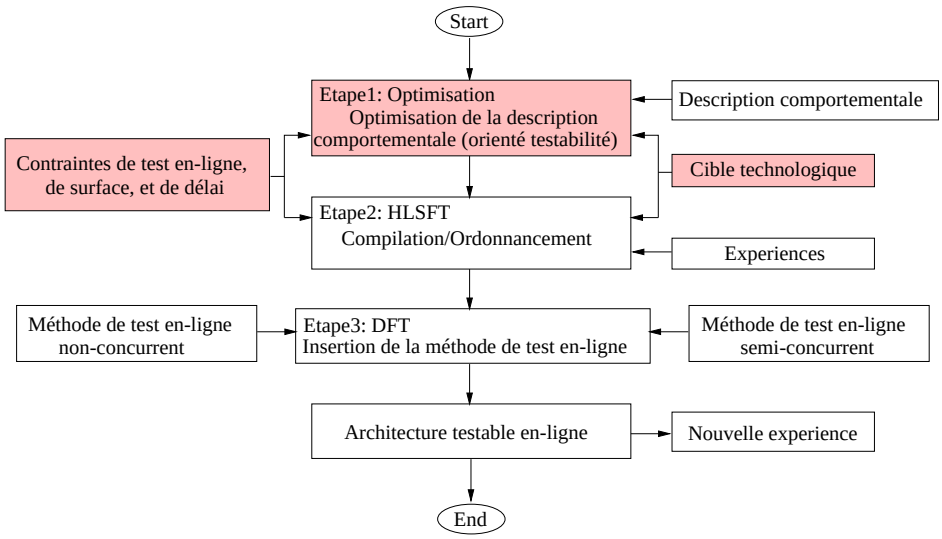


Figure 5.17: *L'intégration de l'optimisation de la description comportementale dans le flot de la synthèse de haut niveau pour le test en-ligne (HLS_OLT).*

Chapitre 6

Compilation et ordonnancement

Dans les chapitres précédents, des méthodes de test en-ligne non-concurrent et semi-concurrent ont été développées. Egalement, une optimisation orientée test en-ligne de la description comportementale a été proposée. A partir d'une description comportementale optimisée, il faut fournir le graphe de flot de données ordonnancé qui répond aux contraintes imposées. Ensuite, une méthode de test en-ligne est intégrée au système au niveau ordonnancement et l'architecture RTL testable en-ligne est générée.

Dans ce chapitre, le problème de la compilation et de l'ordonnancement est adressé. Les contraintes de test en-ligne, en plus de celles de surface et de délai, sont considérées à cette étape. La solution proposée à ce problème est basée sur l'exploration de l'espace de solutions. La méthode se repose sur l'exploration de l'espace de solutions par un algorithme génétique (AG). Une heuristique est utilisée pour la génération de la population initiale. Bien que cette heuristique améliore les performances de l'algorithme génétique, elle ne peut pas garantir la solution optimale.

Cette étude concerne les systèmes de traitement numérique du signal. Les filtres numériques récursifs sont utilisés. Ce type de systèmes possède plusieurs structures de réalisation au niveau de graphe de flot de données. Ces différentes structures seront analysées avant d'aborder le problème de la compilation et de l'ordonnancement.

A partir de chaque structure de réalisation, plusieurs graphes de flots de données ordonnancés sont générés. La testabilité en-ligne pour le test en-ligne non-concurrent et semi-concurrent est évaluée. Les solutions, qui sont les différentes architectures RTL testables en-ligne, sont présentées sous forme génétique et seront explorées par l'algorithme génétique lors de la synthèse d'un nouveau système.

En premier temps, le principe des algorithmes génétiques est exposé. Ensuite, les différentes structures de réalisation des filtres numériques sont étudiées pour permettre l'évaluation de la testabilité en-ligne des différentes architectures possibles au niveau RTL mais aussi pour aider à la génération de la population initiale de l'algorithme génétique. Enfin, l'algorithme de compilation et ordonnancement est établi.

6.1 Terminologie des algorithmes génétiques

Le principe des algorithmes génétiques repose sur deux mécanismes. Ce sont le mécanisme de l'encodage des solutions du problème sous forme de chromosomes et le mécanisme de la fonction d'évaluation de la qualité (*fitness)* de chaque chromosome.

Le mécanisme de l'encodage des solutions peut varier d'un problème à l'autre et d'un algorithme génétique à l'autre. Le chromosome peut être présenté sous forme d'une série de bits mais aussi en utilisant des structures plus complexes telles que les chiffres et les lettres. La technique de la représentation du chromosome dépend du problème. Il n'existe pas une technique générale qui soit la meilleure pour tous les problèmes. Une bonne sélection de la technique d'encodage peut améliorer significativement la qualité de l'algorithme génétique. Une question importante à considérer lors du choix de la technique du codage est : quels sont les facteurs à considérer pour le choix de la technique du codage pour un problème spécifique?

La fonction de l'évaluation de la qualité du chromosome représente le lien entre l'algorithme génétique et le problème à résoudre. Cette fonction analyse le chromosome et rend un nombre ou une liste de nombres qui mesurent les performances du chromosome pour le problème considéré. Elle simule l'interaction de l'individu, sous forme de chromosome, avec son environnement. Cette évaluation détermine le "*fitness*" de l'individu. Le "*fitness*" est le contraire du coût dans les problèmes d'optimisation.

Avec ces éléments, la technique de l'encodage et la fonction de la qualité du chromosome, un algorithme génétique peut être dressé pour simuler l'évolution sur une population initiale de solutions. Cet algorithme suivera les étapes générales suivantes :

1. Initialisation d'une population de chromosomes.
2. Evaluation de chaque chromosome dans la population.
3. Création de nouveaux chromosomes par accouplement (*mating*) des chromosomes actuels.
4. Effacement d'une partie ou de la totalité des chromosomes dans la population actuelle pour faire de la place pour les nouveaux chromosomes.
5. Evaluation des nouveaux chromosomes et insertion des meilleurs chromosomes dans la population.
6. Si le temps est dépassé, arrêter l'algorithme et retourner le meilleur chromosome.

La création de nouveaux chromosomes passe par deux étapes : la sélection des individus et le croisement "*Crossover*" de leurs chromosomes. Ce sont les opérateurs clés pour un algorithme génétique. Ils assurent la survie des meilleurs individus et contrôlent le taux de convergence. Le croisement est l'opérateur principal de la reproduction. Il combine des parties des parents pour générer de nouveaux individus.

La population initiale peut être générée de façon aléatoire, déterministe ou bien donnée par l'utilisateur du programme. Le meilleur individu peut apparaître dans n'importe quelle population. Afin d'accélérer la convergence de l'algorithme vers une solution satisfaisante, une heuristique permettant une génération déterministe de la population initiale est appliquée. Cette heuristique est détaillée au niveau de la génération de la population initiale.

6.1.1 Les opérateurs génétiques

En général, les opérateurs génétiques sont : *selection, crossover, mutation, fitness scaling* et *inversion*. Les deux premiers opérateurs sont les plus importants et ils seront expliqués.

Selection

Plusieurs schémas de sélection ont été proposés mais nous allons nous concentrer sur deux schémas. Ce sont "*Roulette Wheel Selection*" et "*Stochastic Universal Selection*".

Comme le montre la figure (6.1-a), la sélection "*Roulette Wheel*" est proportionnelle à la valeur de "*fitness*" de l'individu. Un individu est sélectionné en tournant la roulette et en marquant la position du marqueur. La probabilité de la sélection d'un individu est donc proportionnelle à sa valeur de "*fitness*".

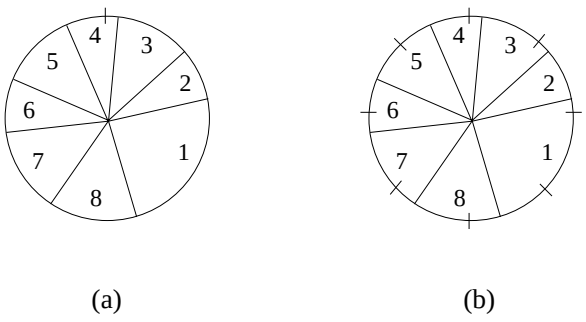


Figure 6.1: *Sélection : Roulette Wheel (a) et Stochastic Universal (b).*

La sélection "*Stochastic Universal*" (figure (6.1-b)), est une version de "*Roulette Wheel*" avec moins de bruit. N marqueurs équidistants sont placés autour de la roulette, où N est le nombre d'individus dans la population. Le nombre de copies sélectionnées d'un individu correspond au nombre de marqueurs qu'il contient après un tour de la roulette.

Crossover

Une fois deux chromosomes sont sélectionnés, l'opérateur de "*Crossover*" est utilisé pour générer une ou plusieurs progénitures (*offsprings*). Les chromosomes parents sont coupés à une position choisie aléatoirement. Les chromosomes des progénitures sont composés des différentes parties des chromosomes des deux parents.

Pour illustration, considérons deux chromosomes parents sous forme de séries de bits (figure (6.2)).

Parent 1 : 1 0 1 1 0 1 1 0 1	*1 1 1 0 0 1 1 0 0*
Parent 2 : 0 0 1 1 0 1 1 0 0	1 0 0 1 0 1 0 0 0
Offspring 1 : 1 0 1 1 0 1 1 0 1	1 0 0 1 0 1 0 0 0
Offspring 2 : 0 0 1 1 0 1 1 0 0	*1 1 1 0 0 1 1 0 0*

Figure 6.2: *One-point crosover.*

Un point est choisi aléatoirement pour couper les deux chromosomes. Les parties sont échangées entre les parents pour donner deux nouvelles pro-génitures.

Cet opérateur peut générer de "mauvais" chromosomes mais ils sont éliminés lors de la sélection pour une nouvelle génération. Il est aussi possible d'ajouter une probabilité de "*Crossover*" de façon à améliorer cet opérateur.

6.1.2 Type et paramètres de l'algorithme génétique

Deux types d'algorithmes génétiques sont utilisés, ce sont l'algorithme simple et l'algorithme "*steady-state*".

Dans un algorithme génétique simple, une génération est complètement remplacée par les nouveaux individus. Dans un "*steady-state*" algorithme seulement une fraction des individus est remplacée par chaque génération.

L'introduction des algorithmes génétiques étant encore nouvelle dans plusieurs domaines pratiques, il n'existe pas un modèle qui peut servir pour tout type de problème. Le choix des algorithmes génétiques pour résoudre un problème implique un travail important d'optimisation du programme. En effet, il est nécessaire de bien choisir le codage des solutions du problème sous forme génétique, la fonction de "*fitness*", les paramètres de l'algorithme génétique et son type.

Un des problèmes importants pour ce type d'algorithmes est le réglage des différents paramètres comme le taux de "*Crossover*", la taille de la population, le nombre de générations qui permet de trouver de bonnes solutions, ...

Une grande importance doit être portée au réglage des paramètres. L'optimisation globale de ces paramètres n'est pas un travail banal. Elle nécessite souvent un algorithme séparé.

Une des solutions possibles à ce problème réside dans l'utilisation d'un algorithme génétique de type "*meta-level*". Ce type consiste à utiliser deux niveaux d'algorithmes génétiques. Le premier niveau adresse directement le problème à résoudre et le deuxième niveau, dit "*meta-level*", contrôle les paramètres du premier niveau pour l'optimiser.

Une autre solution consiste à combiner l'approche génétique avec une approche déterministe. Il est possible de choisir soit une approche traditionnelle telle que la programmation linéaire qui garantit une solution optimale, soit une heuristique développée spécifiquement pour le problème à résoudre. Les heuristiques permettent la génération d'une solution satisfaisante en temps raisonnable.

Notre introduction des algorithmes génétiques dans le domaine de la synthèse de haut niveau pour le test en-ligne considère le type "*steady-state*". L'approche consiste à combiner l'approche génétique avec une heuristique. Cette heuristique est introduite au niveau de la génération de la population initiale et sert à accélérer la convergance de l'algorithme vers une solution satisfaisante. Le problème de réglage des paramètres peut être résolu par l'évaluation de la qualité de l'algorithme par expérimentation.

6.2 Les filtres numériques récursifs IIR

Le choix des filtres numériques récursifs de type IIR pour l'implémentation de la méthode est justifié par les points suivants :

- Ils représentent un cas général des filtres numériques. Ce qui fait de l'extension de l'application de la méthode pour les filtres FIR une tâche facile.

- Ils ont des problèmes d'instabilité et nécessitent des structures spécifiques pour la réalisation. Ce qui permet une généralisation de la méthode.

- Ils conviennent pour l'implémentation directe des filtres analogiques et pour les applications à débit élevé. Plus particulièrement les filtres elliptiques qui donnent moins de coefficients que les filtres FIR. Cela représente un avantage pour la testabilité en-ligne du filtre. Moins de coefficients implique moins d'opérations à réaliser et donc une période de repos plus importante.

Pour illustrer le dernier point, considérons les deux fonctions de transfert suivantes qui représentent une réponse amplitude-fréquence identique[48] :

$$H_1(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

où :

$a_0 = 0.4981819$, $a_1 = 0.9274777$, $a_2 = 0.4981819$ b_1 = -0.6744878, b_2 = -0.3633482

et

$$H_2(z) = \sum_{k=0}^{11} h(k) z^{-k}$$

où :

$$\begin{array}{llll} h(0) = 0.54603280 \times 10^{-2} & = h(11) & h(3) = -0.55384370 \times 10^{-1} & = h(8) \\ h(1) = -0.45068750 \times 10^{-1} & = h(10) & h(4) = -0.63428410 \times 10^{-1} & = h(7) \\ h(2) = 0.69169420 \times 10^{-1} & = h(9) & h(5) = 0.57892400 \times 10^{0} & = h(6) \end{array}$$

$H_1(z)$ est un filtre IIR réalisé par la structure dans la figure (6.3-a). $H_2(z)$ est un filtre FIR réalisé par la structure dans la figure (6.3-b).

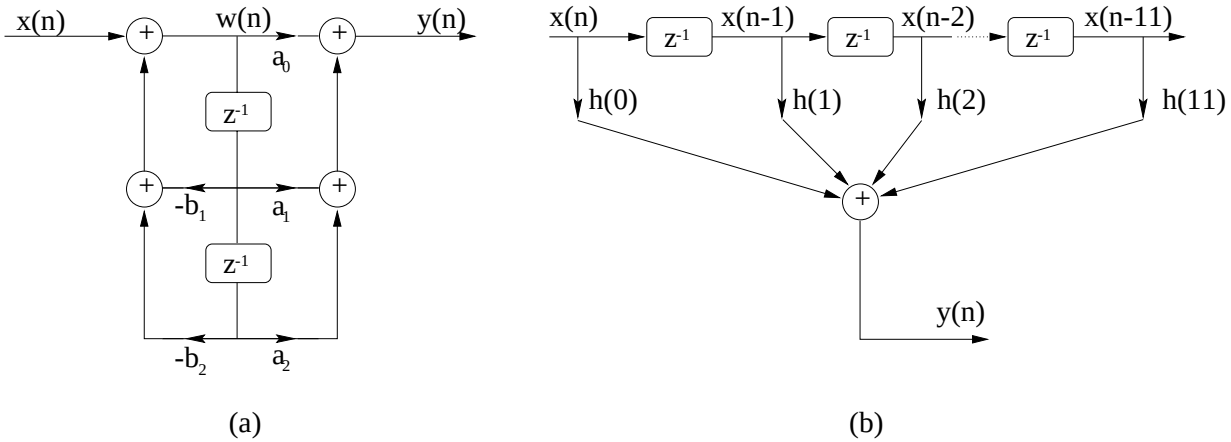


Figure 6.3: *Structures de réalisation des fonctions de transfert $H_1(z)$ (a) et $H_2(z)$ (b).*

On peut constater la différence entre les deux structures :

Filtre	FIR	IIR
Nombre de multiplications	12	5
Nombre d'additions	11	4
Eléments de stockage	24	8

6.3 Structures de réalisation

Les structures traitées par cette section sont les différentes structures des graphes de flot de données qui implémentent la fonction de transfert du filtre.

Plusieurs structures existent pour réaliser les filtres IIR. La première structure est celle qui représente une réalisation directe de l'équation :

$$y(n) = \sum_{i=0}^{N} a_l x(n-i) - \sum_{i=1}^{N} b_i y(n-i)$$

Cette structure est appelée la forme directe 1 (figure (6.4)). Elle est très sensitive à l'effet de la limitation du nombre de bits des coefficients et donc évitée dans beaucoup des applications pratiques.

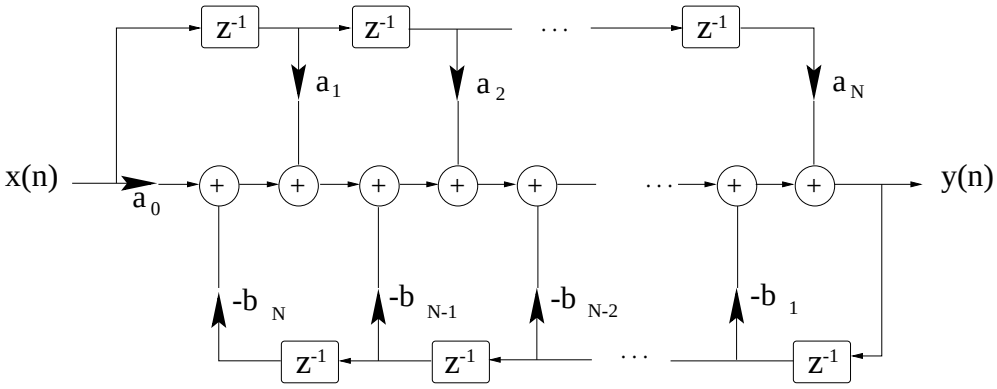


Figure 6.4: *Forme directe 1.*

La deuxième structure est appelée la forme directe 2 (figure (6.5)). Pour obtenir cette structure, la fonction de transfert peut être écrite de la façon suivante :

$$H(z) = \frac{Y(z)}{X(z)} = \underbrace{\left(\frac{1}{\sum_{i=0}^{N} b_i z^{-1}}\right)}_{H_1(z)} \underbrace{\left(\sum_{i=0}^{N} a_i z^{-i}\right)}_{H_2(z)}$$

Cette forme permet de réaliser le filtre comme une cascade de deux filtres dont les fonctions de transfert sont $H_1(z)$ et $H_2(z)$. Cette structure est appelée aussi la forme canonique car elle représente le nombre minimum d'éléments de mémorisation.

Deux autres structures existent. Ce sont les structures décomposées. Au lieu de réaliser $H(z)$ directement, on peut effectuer une décomposition en somme ou en produit de fonctions élémentaires du premier ou de second ordre réalisés séparément.

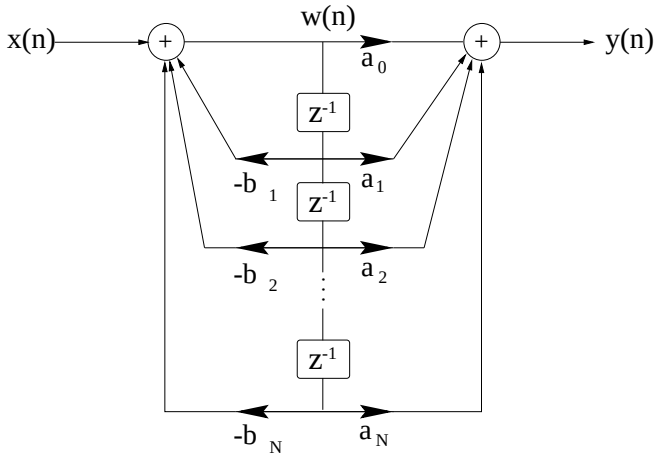


Figure 6.5: *Forme directe 2.*

La décomposition en produit correspond à la structure cascade (figure (6.6-a)), où le filtre est réalisé par une suite de K cellules élémentaires du premier et de second ordre :

$$H(z) = \frac{Y(z)}{X(z)} = a_0 \Pi_{i=1}^{K} H_i(z)$$

où $H_i(z)$ est, soit une section de second ordre :

$$H_i(z) = \frac{1 + a_{1i} z^{-1} + a_{2i} z^{-2}}{1 + b_{1i} z^{-1} + b_{2i} z^{-2}}$$

soit une section du premier ordre :

$$H_i(z) = \frac{1 + a_{1i} z^{-1}}{1 + b_{1i} z^{-1}}$$

et K est la valeur entière de $(N+1)/2$.

Cette structure est la plus utilisée car elle représente, en plus de sa modularité des caractéristiques avantageuses de faible sensibilité aux arrondis des coefficients et au bruit de calcul.

La décomposition en somme correspond à la structure parallèle (figure (6.6-b)), où le filtre est réalisé par la mise en parallèle de K cellules élémentaires du premier et de second ordre :

$$H(z) = \frac{Y(z)}{X(z)} = C + \sum_{i=1}^{K} H_i(z)$$

où $H_i(z)$ est, soit une section de second ordre :

$$H_i(z) = \frac{a_{0i} + a_{1i} z^{-1}}{1 + b_{1i} z^{-1} + b_{2i} z^{-2}}$$

soit une section du premier ordre :

$$H_i(z) = \frac{a_{0i}}{1 + b_{1i}z^{-1}}$$

et K est la valeur entière de $(N+1)/2$, $C = a_N/b_N$.

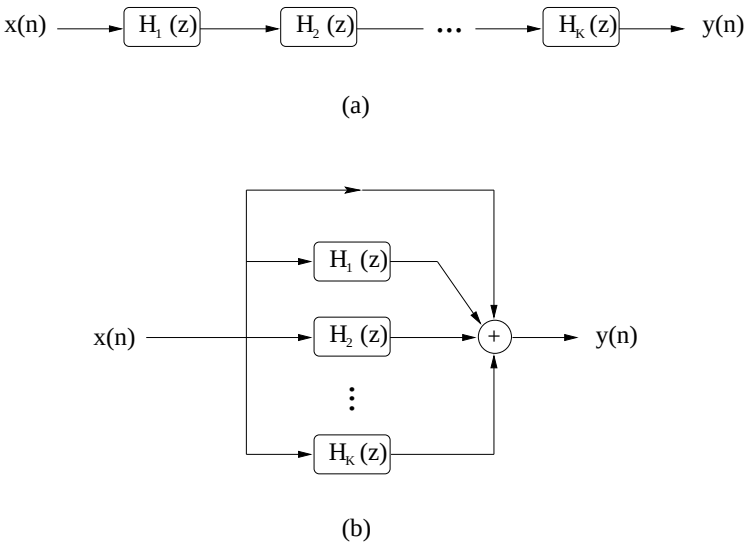


Figure 6.6: *Les structures décomposées, la structure cascade (a) et la structure parallèle (b).*

Ces quatre structures sont les plus utilisées pour la réalisation des filtres numériques. Il existe d'autres structures mais elles sont connues dans des domaines spécifiques d'applications.

Dans la section suivante, la testabilité en-ligne de ces différentes structures sera évaluée. Cette évaluation est basée sur l'analyse de la corrélation entre la fréquence opérationnelle, la fréquence d'échantillonnage et l'architecture du filtre au niveau RTL.

6.4 Evaluation de la testabilité en-ligne : test non-concurrent

L'avancement technologique, dans le domaine de la fabrication des circuits intégrés, permet aux systèmes numériques de disposer d'une période plus importante de temps de repos. Etant donné le nombre limité de vecteurs de test à appliquer sur une unité fonctionnelle (figure (6.7) cas du multiplieur et de l'additionneur), l'application de la méthode de test en-ligne non-concurrent aux nouveaux systèmes de traitement numérique de signal permet d'obtenir de bons résultats en testabilité en-ligne.

La testabilité en-ligne des différentes structures de réalisation étudiées dans la section précédente est évaluée dans cette section. La méthode de test en-ligne non-concurrent est adressée. Cette évaluation sera effectuée par l'analyse, pour chaque structure de réalisation, de la corrélation entre la fréquence opérationnelle du filtre, la fréquence d'échantillonnage et la largeur en bit des unités fonctionnelles des différentes architectures au niveau RTL. Etant donnée la fréquence d'échantillonnage, le nombre disponible des unités fonctionnelles et la largeur en bit de ces unités, la fréquence opérationnelle, qui permet une latence de faute

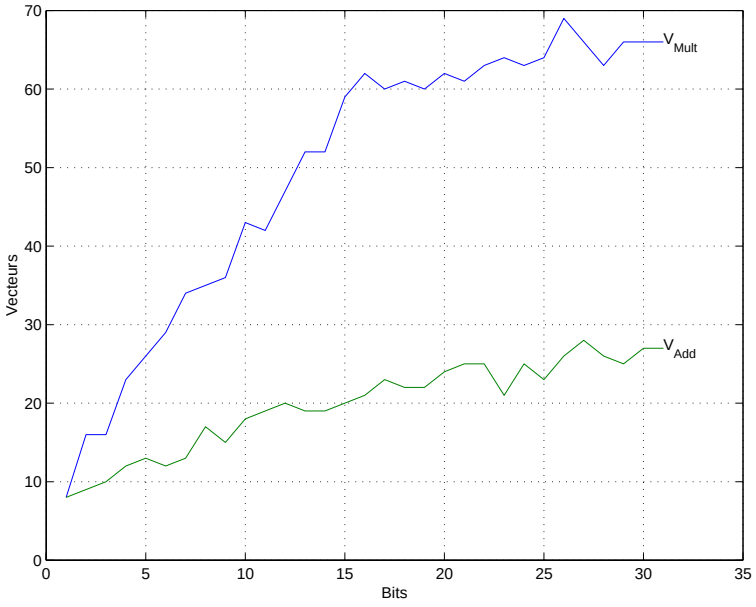


Figure 6.7: *Le nombre de vecteurs de test pour le multiplieur et l'additionneur obtenus par l'outil design_analyzer de SYNOPSYS pour la technologie 0.6 µm, cub de AMS.*

égale à 1, est calculée. La latence de faute est égale à 1 quand tous les vecteurs de test peuvent être appliqués aux unités fonctionnelles pendant une période d'échantillonnage. Le cas du multiplieur est considéré car il possède le nombre le plus élevé de vecteurs de test. Cette étude permet la définition de l'architecture, au niveau RTL, qui permet d'atteindre un niveau déterminé de testabilité en-ligne. Cette définition de l'architecture du filtre au niveau RTL comprend le nombre de pas de contrôle du graphe de flot de données ordonnancé, le nombre d'opérateurs au niveau RTL et le choix de la cible technologique.

Pour réaliser cette évaluation plusieurs éléments sont à déterminer. D'abord, l'architecture ciblée au niveau RTL est sélectionnée. Pour illustration, nous considérons deux types d'architectures. La première architecture est celle qui comporte un seul multiplieur et un seul additionneur. Elle représente la réalisation en série du graphe de flot de données et permet l'implémentation d'une unité de MAC (Multiply and ACumulate) pour un DSP. La deuxième architecture est l'architecture parallèle qui comporte d'autant de multiplieurs que nécessaire. Elle réalise l'ordonnancement ASAP (As Soon As Possible) du graphe de flot de données avec le nombre maximum possible de multiplieurs.

Ensuite, le nombre de pas de contrôle de la période fonctionnelle de chaque structure est évalué selon l'architecture ciblée. Ce nombre varie selon l'ordre du filtre N et le nombre des opérateurs disponibles au niveau RTL.

Enfin, le nombre de vecteurs de test est ajouté au nombre de pas de contrôle de la période fonctionnelle et la fréquence opérationnelle est calculée. L'évaluation de la testabilité en-ligne se fait, donc, par les étapes suivantes :

- Détermination du nombre maximal de vecteurs de test à appliquer sur les différents

opérateurs. Le nombre de vecteurs de test du multiplieur (V_{Mult}) est considéré.

- Détermination de la structure de réalisation au niveau graphe de flot de données.
- Détermination de l'architecture du filtre au niveau RTL.
- Calcul du nombre de pas de contrôle (N_{ctrl}) de la période fonctionnelle T_f.
- Pour une fréquence d'échantillonnage donnée (F_e), calcul de la fréquence opérationnelle (F_{Op}) qui permet d'atteindre une testabilité en-ligne égale ou supérieure à 1 :

$$F_{Op} = F_e(N_{ctrl} + V_{Mult})$$

Pour illustration, l'ordonnancement de l'architecture série et celle parallèle sont réalisées pour un filtre IIR elliptique de second ordre (figure (6.8)). La forme directe 1 est utilisée.

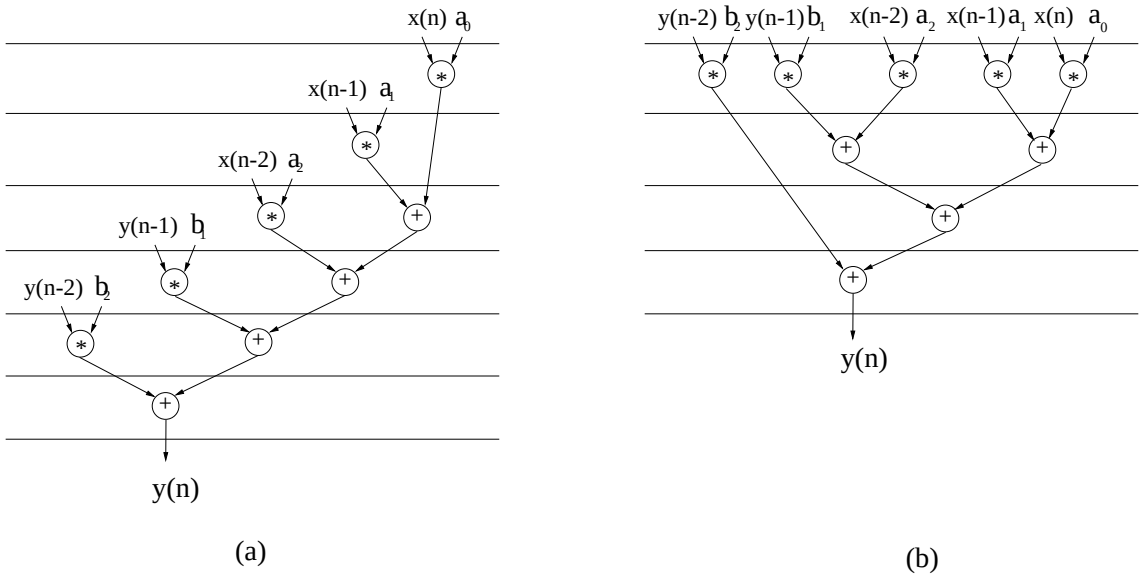


Figure 6.8: *L'ordonnancement de l'architecture série (a) et parallèle (b). Filtre IIR* 2^{ed} *ordre.*

6.4.1 Formes directes 1 et 2

Une telle forme d'un filtre IIR d'ordre N contient $2N + 1$ opérations de multiplication. Pour l'architecture parallèle, ces $2N + 1$ opérations de multiplication peuvent être ordonnancées en premier pas de contrôle. Ensuite, $2N$ opérations d'addition calculent la somme des produits en K pas de contrôle où K est la valeur entière de : $log_2(2N + 1)$. En total, il faut $K + 1$ pas de contrôle pour cette architecture. L'équation qui permet le calcul de la fréquence opérationnelle nécessaire pour atteindre une latence de faute égale à 1 devient :

$$F_{Op} = F_e(K + 1 + V_{Mult})$$

Pour l'architecture série, il faut $2N + 1$ pas de contrôle pour réaliser les opérations de multiplication et un pas de contrôle supplémentaire pour la dernière opération d'addition. En total, il faut $2N + 2$ pas de contrôle pour cette architecture. Cela donne :

$$F_{Op} = F_e(2N + 2 + V_{Mult})$$

L'évaluation du nombre de pas de contrôle nécessaires pour les formes directes 1 et 2 (figure (6.9)), est faite pour un ordre du filtre qui varie de N= 1 jusqu'à N= 10.

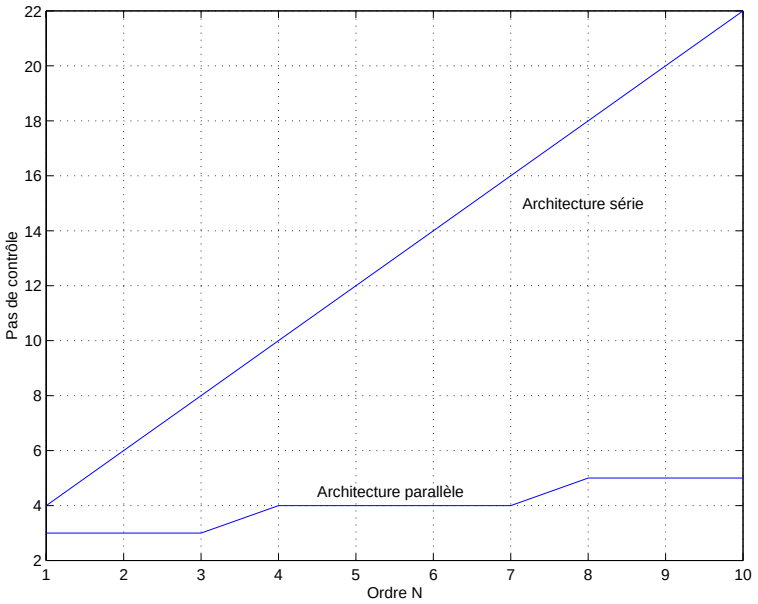


Figure 6.9: *Le nombre de pas de contrôle par rapport à l'ordre du filtre.*

Le nombre de vecteurs de test par rapport à la largeur du multiplieur (figure (6.7)), ainsi que le nombre de pas de contrôle par rapport à l'ordre du filtre (figure (6.9)), sont utilisés pour illustrer l'effet de la fréquence opérationnelle sur la testabilité en-ligne de ces différentes architectures.

Nous considérons un filtre IIR elliptique d'ordre variant de N= 2 à 10 réalisé en architecture série (figure (6.10)), et en architecture parallèle (figure (6.11)).

6.4.2 Forme décomposée

L'analyse de la testabilité en-ligne de la forme décomposée est basée sur la même analyse pour une section de second ordre (figure (6.12-a)).

Cette section contient cinq opérations de multiplication. Pour illustration, considérons aussi l'architecture parallèle et celle série au niveau RTL.

L'architecture parallèle contient cinq opérations de multiplication ordonnancées en premier pas de contrôle. La somme de ces produits se fait par quatre opérations d'addition en deux pas de contrôle (figure (6.12-b)).

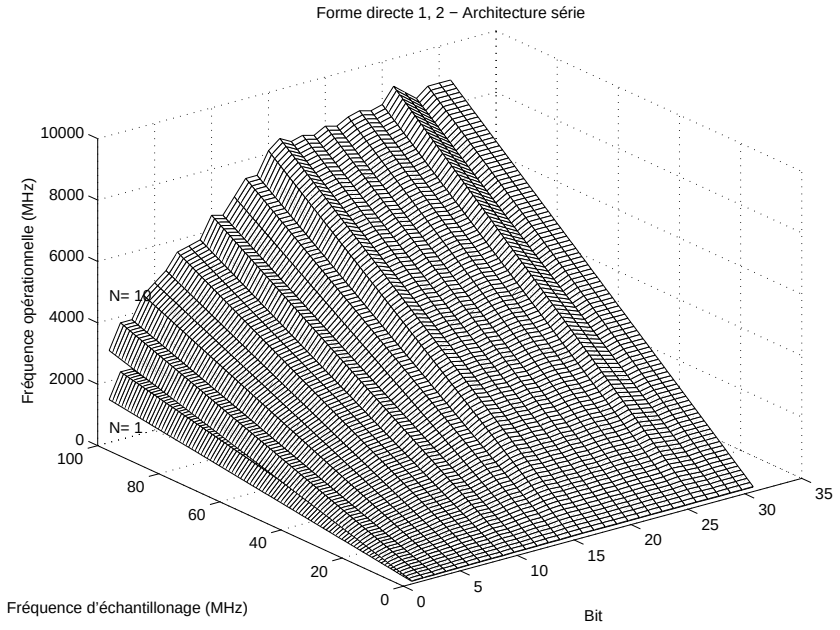


Figure 6.10: *Evaluation de la testabilité en-ligne de l'architecture série pour les formes directes 1 et 2 et pour ordre de filtre N= 2 à 10. Test non-concurrent.*

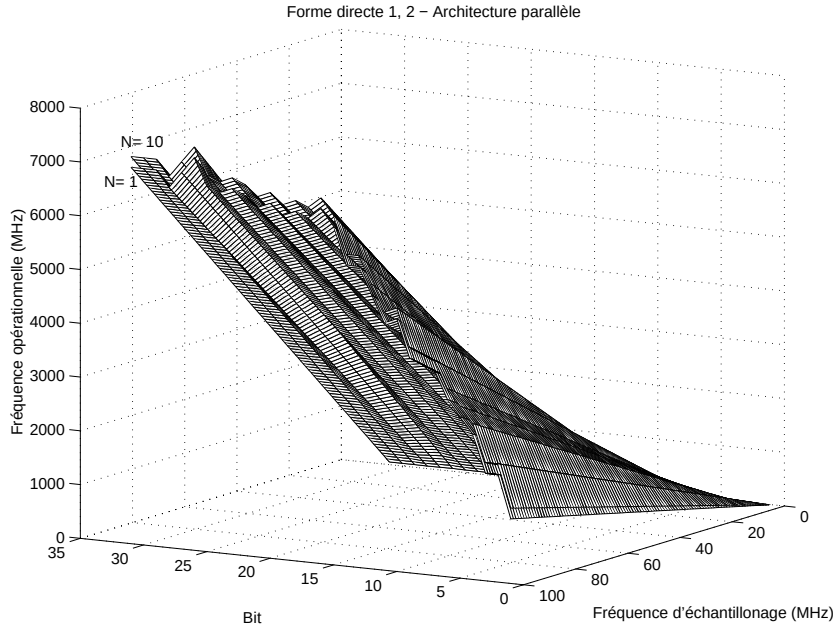


Figure 6.11: *Evaluation de la testabilité en-ligne de l'architecture parallèle pour les formes directes 1 et 2 et pour ordre de filtre N= 2 à 10. Test non-concurrent.*

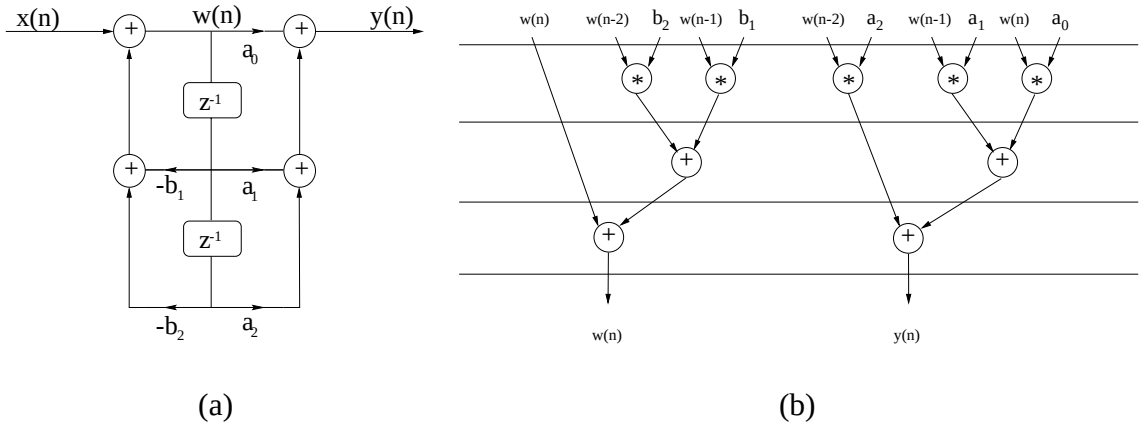


Figure 6.12: *Une section de second ordre réalisée par la forme directe 2.*

Structure cascade

Un filtre IIR d'ordre $N \geq 2$ peut être décomposé en k sections de second ordre mises en cascade, k étant la valeur entière de $N/2$. On peut constater de la figure (6.12-b) que deux pas de contrôle sont des temps de repos pour la multiplication. Deux vecteurs de test peuvent être insérés donc dans l'ordonnancement de chaque section de second ordre. En total, $2k$ vecteurs de test sont appliqués au système. Pour compléter l'application du reste des vecteurs de test il faut $V_{Mult} - 2k$ pas de contrôle. Le nombre total de pas de contrôle nécessaires pour l'architecture qui réalise la structure cascade est $V_{Mult} + k$. La fréquence opérationnelle qui garantit une latence de faute égale à 1 est dans ce cas :

$$F_{Op} = F_e(V_{Mult} + k)$$

Structure parallèle

Les k sections, mises en parallèle, nécessitent 3 pas de contrôle pour réaliser le calcul de chaque section suivis par K pas de contrôle pour calculer la somme des différentes sections, K est la valeur entière de : $log_2 k$.

Deux vecteurs de test peuvent être appliqués dans les deux derniers pas de contrôle de l'ordonnancement de chaque section et K vecteurs dans la période du calcul de la somme. Le nombre total de pas de contrôle, nécessaire pour atteindre une testabilité égale à 1, s'évalue donc à $V_{Mult} - 2 - K + K + 3 = V_{Mult} + 1$. Cela donne une fréquence opérationnelle F_{Op} égale à :

$$F_{Op} = F_e(V_{Mult} + 1)$$

La structure cascade et celle parallèle conduisent à un niveau pareille en performance pour le même niveau en testabilité en-ligne. La réalisation de ces structures conduit à une corrélation entre la fréquence opérationnelle, la fréquence d'échantillonnage et la largeur en bit du multiplieur comparable à celle des formes directes 1 et 2 selon l'architecture choisie. On peut considérer la figure (6.10) pour la réalisation des sections en architecture série et la figure (6.11) pour la réalisation des sections en architecture parallèle.

6.4.3 Equations d'état

Un filtre numérique peut être représenté par ses équations d'état. Cette représentation intéresse certaines méthodes de test en-ligne concurrent telle que la méthode développée par Abdelhay [2]. L'évaluation du coût de la réalisation du filtre à partir de ses équations d'état nous intéresse pour deux raisons. La première est la comparaison avec les méthodes de test en-ligne non-concurrent et semi-concurrent proposées par la présente étude. La deuxième est l'inclusion de la méthode présentée dans [2] dans un outil général de synthèse de haut niveau pour le test en-ligne.

Ce type de représentation concerne les systèmes digitaux linéaires. Un tel système peut être représenté par ses équations d'état :

$$x(t+1) = A.x(t) + B.u(t)$$

$$y(t) = C.x(t) + D.u(t)$$

Les matrices A, B, C et D sont de dimensions appropriées. Pour un filtre IIR d'ordre N, la matrice A contient N colonnes et N lignes ($N \times N$) alors que la matrice B contient N lignes et une seule colonne ($N \times 1$). La matrice C contient N colonnes et une seule ligne ($1 \times N$) et la matrice D contient toujours un seul élément (1×1).

Comme pour les formes directes 1 et 2, considérons la réalisation des équations d'état d'un filtre par deux architectures, l'architecture série et celle parallèle. En plus, une architecture intermédiaire est considérée. Cette troixième architecture contient autant de multiplieurs que de multiplications nécessaires pour le calcul d'un seul état $x_i(t)$.

Pour les équations d'état, un filtre IIR d'ordre N contient $(N+1)^2$ opérations de multiplication. Pour l'architecture parallèle, $(N+1)^2$ opérations de multiplication peuvent être ordonnancées en premier pas de contrôle et $(N+1)^2 - 1$ opérations d'addition calculent la somme des produits en K_p pas de contrôle où K_p est la valeur entière de : $log_2(N+1)^2$. En total, il faut $K_p + 1$ pas de contrôle pour cette architecture. L'équation du calcul de la fréquence opérationnelle correspondant à une latence de faute égale à 1 est donc :

$$F_{Op} = F_e(K_p + 1 + V_{Mult})$$

Pour l'architecture série, il faut $(N+1)^2$ pas de contrôle pour réaliser les opérations de multiplication et un pas de contrôle supplémentaire pour la dernière opération d'addition. En total, il faut $(N+1)^2 + 1$ pas de contrôle pour cette architecture.

$$F_{Op} = F_e((N+1)^2 + 1 + V_{Mult})$$

Pour l'architecture intermédiaire, il faut N+1 opérations de multiplication pour réaliser le calcul d'un état et le calcul de la sortie. Donc, N+1 opérations de multiplication peuvent être ordonnancées en un pas de contrôle et k_i pas de contrôle pour calculer la somme des produits où k_i est la valeur entière de : $log_2(N+1)$. En total, il faut $k_i + 1$ pas de contrôle pour le calcul d'un état et pour le calcul de la sortie. Il y a N états et une sortie ce qui fait $N + k_i + 1$ pas de contrôle en total. Cela donne :

$$F_{Op} = F_e(N + k_i + 1 + V_{Mult})$$

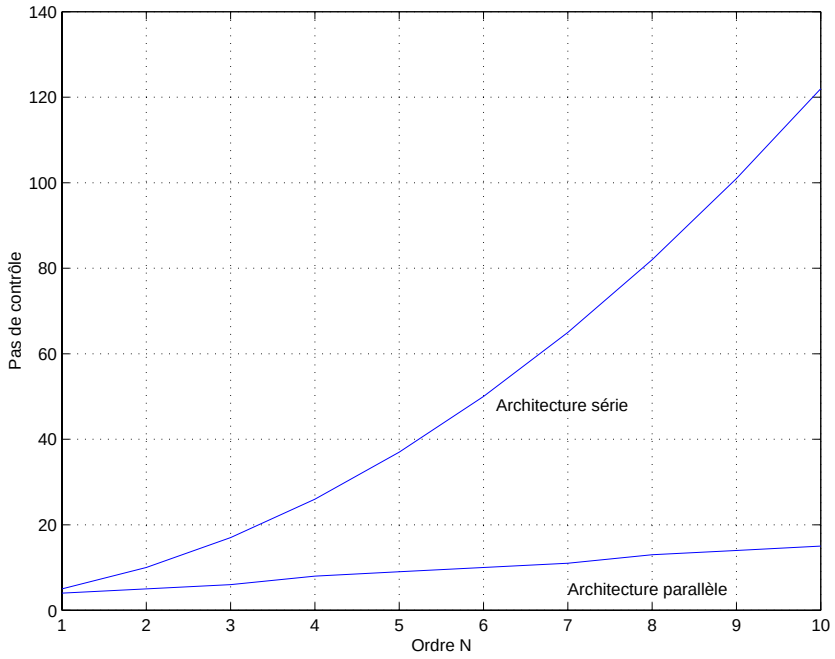


Figure 6.13: *Le nombre de pas de contrôle par rapport à l'ordre du filtre.*

Le nombre de pas de contrôle nécessaires pour réaliser ces architectures est présenté dans la figure (6.13) pour N= 1 à N= 10.

Le nombre de vecteurs de test par rapport au largeur du multiplieur (figure (6.7)), ainsi que le nombre de pas de contrôle par rapport à l'ordre du filtre (figure (6.13)), sont utilisés pour illustrer l'effet de la fréquence opérationnelle sur la testabilité en-ligne de ces différentes architectures.

Considérons aussi un filtre IIR elliptique d'ordre variant de N= 2 à 10 réalisé en architecture série (figure (6.14)), en architecture parallèle (figure (6.15)) et en architecture intermédiaire (figure (6.16)). A rappeler que le calcul de la fréquence opérationnelle est basé sur l'idée : étant donné la largeur en bit du multiplieur et la fréquence d'échantillonnage, quelle est la fréquence opérationnelle qui permet une testabilité en-lign égale à 1.

6.5 Evaluation de la testabilité en-ligne : test semi-concurrent

Le test en-ligne semi-concurrent consiste à insérer, dans l'ordonnancement nominal du système, un graphe de test en-ligne. Ce graphe réalise le même calcul réalisé par le graphe nominal pour comparer les résultats.

Pour avoir une latence de faute égale à 1, le graphe de test en-ligne doit terminer son calcul avant l'arrivée de l'échantillon suivant. Cela veut dire que la fréquence opérationnelle nécessaire pour obtenir une telle testabilité doit permettre l'insertion totale du graphe de test en-ligne dans la période fonctionnelle et la période de repos. Pour cela, la fréquence

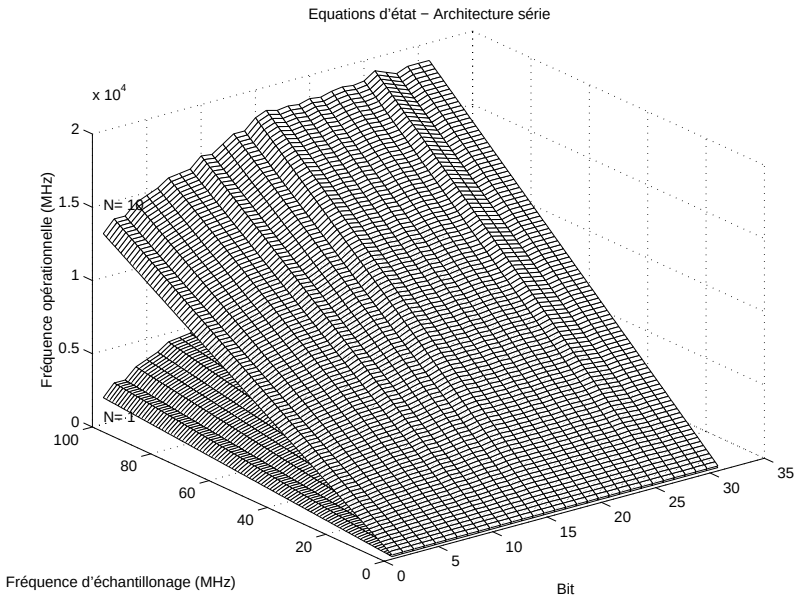


Figure 6.14: *Evaluation de la testabilité en-ligne de l'architecture série à partir des équations d'état et pour ordre N= 2 à 10. Test non-concurrent.*

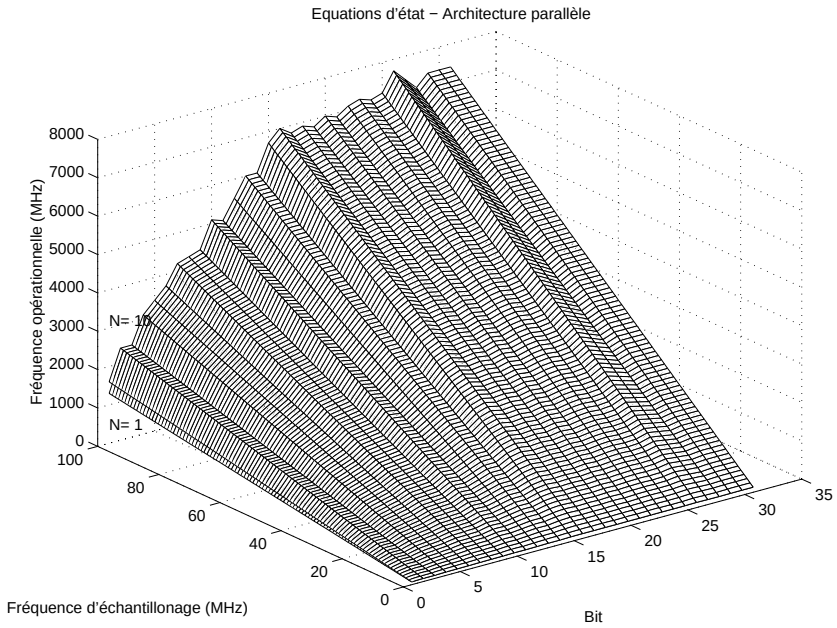


Figure 6.15: *Evaluation de la testabilité en-ligne de l'architecture parallèle à partir des équations d'état et pour ordre N= 2 à 10. Test non-concurrent.*

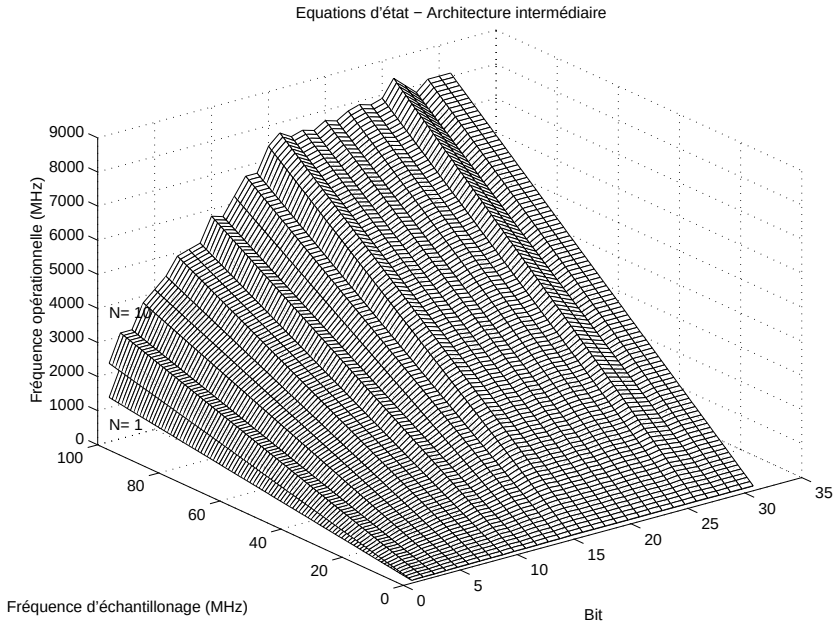


Figure 6.16: *Evaluation de la testabilité en-ligne de l'architecture intermédiaire à partir des équations d'état et pour ordre N= 2 à 10. Test non-concurrent.*

opérationnelle doit satisfaire à l'équation suivante :

$$F_{Op} = F_e(N_{ctrl} + N_{test} - N_{Idle})$$

où :

N_{ctrl} : le nombre de pas de contrôle de la période fonctionnelle nominale ;

N_{test} : le nombre de pas de contrôle de la période de test en-ligne ;

N_{Idle} : le nombre des temps oisifs disponibles dans la période fonctionnelle nominale et qui peuvent être exploités pour le graphe de test en-ligne.

Nous allons analyser la corrélation entre la fréquence opérationnelle, la fréquence d'échantillonnage et l'ordre du filtre IIR. Nous considérons, comme pour la méthode de test en-ligne non-concurrent, deux architecture de réalisation au niveau RTL ; ce sont l'architecture série et l'architecture parallèle. Le calcul de la fréquence opérationnelle qui se fait par les étapes suivantes :

- Détermination de l'architecture ciblée au niveau RTL (architecture parallèle ou série).
- Pour un ordre N donné, calcul du nombre de pas de contrôle N_{ctrl} de la période fonctionnelle du filtre.
- Evaluation des temps oisifs dans la période fonctionnelle et calcul de N_{Idle}.
- Calcul de la fréquence opérationnelle F_{Op}.

6.5.1 Formes directes 1 et 2

Pour l'architecture série, les 2N+1 opérations de multiplication sont réparties sur le même nombre de pas de contrôle. Le dernier pas de contrôle contient seulement une opération d'addition. Le graphe de test en-ligne peut, dans ce cas, être ordonnancé à partir du dernier pas de contrôle. Il faut, donc, 2N+2 pas de contrôle pour le graphe nominal et 2N+1 pas de contrôle pour le graphe de test en-ligne. La fréquence opérationnelle est donc donnée par l'équation :

$$F_{Op} = F_e(4N + 3)$$

Pour l'architecture parallèle, le premier pas de contrôle contient les 2N+1 opérations de multiplication. Les K pas de contrôle de l'addition, où $2N+1 \leq 2^K$, peuvent être utilisés pour ordonnancer le graphe de test en-ligne. Le pas de contrôle du début de l'ordonnancement du graphe de test en-ligne dépend des contraintes de surface et de délai. L'ordonnancement du graphe de test en-ligne dès le deuxième pas de contrôle implique une dégradation en surface en raison de l'utilisation des additionneurs supplémentaires mais améliore la testabilité en-ligne. Nous supposons, pour cette évaluation que les contraintes de surface permettent l'ajout des additionneurs supplémentaires. Dans ce cas, un seul pas de contrôle sera ajouté à la fin de la période fonctionnelle pour terminer le calcul du graphe de test en-ligne. La fréquence opérationnelle peut être calculée par l'équation :

$$F_{Op} = F_e(K + 2)$$

Figure (6.17) représentent l'évaluation de la corrélation entre la fréquence d'échantillonnage, la fréquence opérationnelle et l'ordre du filtre pour les deux architectures.

Pour la forme directe 2, le même raisonnement peut être appliqué pour l'architecture série. Cependant, l'architecture parallèle représente une petite différence.

Le nombre des opérations de multiplication reste 2N+1 dont N pour $H_1(z)$ et N+1 pour $H_2(z)$. Il y a, donc, N opération d'addition pour les N+1 opérations de multiplication. Le calcul de la somme des produits de $H_2(z)$ prend K pas de contrôle où $N + 1 \leq 2^K$. La différence entre les formes directes 1 et 2 est essentiellement dans la valeur de K. Cependant, cette différence en nombre de pas de contrôle n'entraîne pas une modification importante en testabilité en-ligne comme le montrent figure (6.17).

6.5.2 Forme décomposée

Pour la réalisation de la forme décomposée, nous considérons l'utilisation des sections de second ordre réalisées en forme directe 2 (figure (6.12)). Pour chaque section de second ordre, il faut trois pas de contrôle pour l'ordonnancement du graphe nominal et un pas de contrôle supplémentaire pour compléter l'ordonnancement du graphe de test en-ligne. Si le filtre est composé de k sections, k pas de contrôle supplémentaires sont introduits à l'ordonnancement nominal. Nous considérons aussi deux types d'architectues.

La première architecture est celle qui dispose, en matériel, d'une seule section de second ordre pour réaliser toute la structure, qu'elle soit cascade ou parallèle. Dans ce cas, il faut introduire k pas de contrôle à l'architecture nominale et l'équation de l'évaluation de la testabilité en-ligne devient :

$$F_{Op} = 4kF_e$$

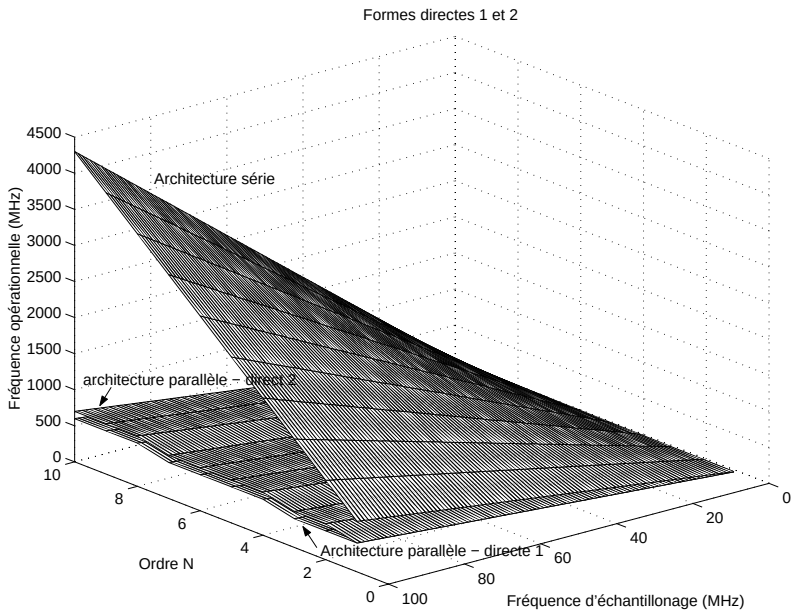


Figure 6.17: *Evaluation de la testabilité en-ligne des formes directes 1 et 2 et pour ordre de filtre N= 2 à 10. Test semi-concurrent.*

La deuxième architecture est celle qui dispose, en matériel, d'autant de section que nécessaire. Dans ce cas, seulement un pas de contrôle est ajouté à l'ordonnancement de chaque section de second ordre et l'équation devient :

$$F_{Op} = 4F_e$$

L'évaluation de la testabilité en-ligne des formes factorisées est illustrée dans la figure (6.18).

6.5.3 Equations d'état

Pour les équations d'état, considérons encore les trois types d'architectures qui ont été utilisées pour l'évaluation de la méthode de test en-ligne non-concurrent.

La première architecture, l'architecture parallèle, nécessite l'ajout d'un pas de contrôle supplémentaire pour le graphe de test en-ligne. L'équation de la fréquence opérationnelle, qui garantit une latence de faute égale à 1, est :

$$F_{Op} = F_e(K_p + 2)$$

La deuxième architecture, l'architecture série, nécessite l'ajout de $2(N+1)^2+1$ pas de contrôle pour l'ordonnancement du graphe de test en-ligne. L'équation de la fréquence opérationnelle devient :

$$F_{Op} = F_e(2(N+1)^2 + 1)$$

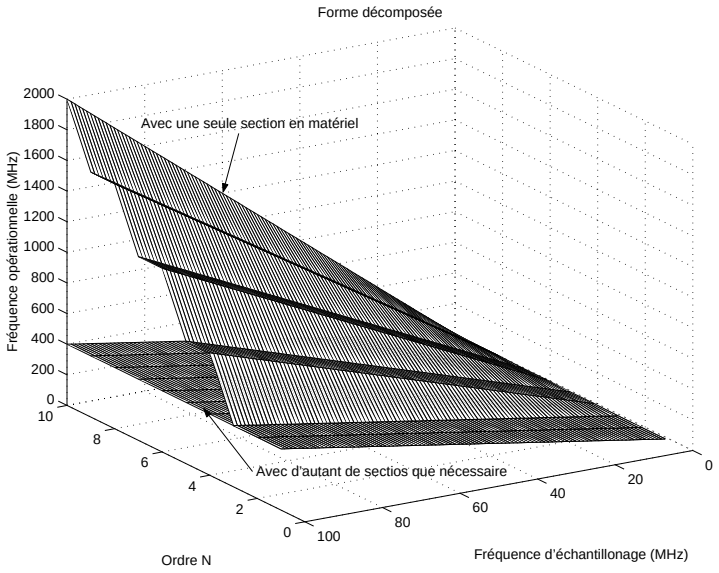


Figure 6.18: *Evaluation de la testabilité en-ligne de la forme décomposée et pour ordre de filtre N= 2 à 10. Test semi-concurrent.*

La troisième architecture, l'architecture intermédiaire, nécessite l'ajout de N+1 pas de contrôle supplémentaires. A savoir que le graphe de test en-ligne peut être ordonnancé à partire du pas de contrôle $N + 2$ de l'ordonnancement nominal. Cela, aussi, dépend des contraintes de surface au niveau du nombre d'additionneurs disponibles pour l'architecture au niveau RTL. Dans le cas où ces contraintes de surface permettent l'ordonnancement du graphe de test en-ligne dès le pas de contrôle $N + 2$, l'équation de la fréquence opérationnelle devient :

$$F_{Op} = F_e(2(N + 1) + K_i)$$

L'évaluation de la testabilité en-ligne pour ces trois architectures est présentée dans la figure (6.19).

6.6 Comparaison et évaluation des différentes structures

L'évaluation de la testabilité en-ligne des différentes structures des filtres IIR, montre que les structures décomposées représentent des caractéristiques avantageuses en performance des architectures testables en-ligne. En plus, ce type de structures est le plus utilisé pour la réalisation des filtres IIR.

Pour obtenir une architecture RTL testable en-ligne avec les meilleurs performances possibles, il est conseillé d'utiliser une structure décomposée avec la méthode de test en-ligne semi-concurrent (figure (6.18)).

Si les formes directes 1 et 2 sont utilisées ou si le filtre est synthétisé à partir de ses équations d'état, les architectures parallèles sont conseillées pour les meilleures performances.

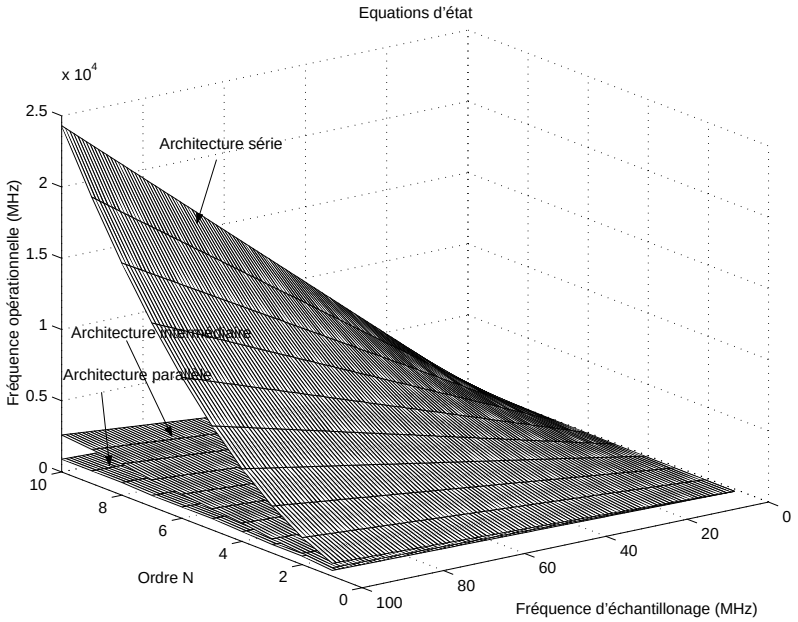


Figure 6.19: *Evaluation de la testabilité en-ligne des architectures réalisées à partir des équations d'état et pour ordre N= 2 à 10. Test semi-concurrent.*

Le même raisonnement peut être appliqué aux filtres de type FIR pour les formes directes 1 et 2 et les équations d'état.

6.7 L'espace de solutions

L'espace de solutions pour les filtres IIR comprend toutes les architectures, au niveau RTL, qui peuvent résulter des différentes structures de réalisation. En plus de la structure de réalisation, nous allons considérer le nombre de multiplieurs au niveau RTL comme l'élément qui distingue une architecture de l'autre. Sans perte de généralité, l'analyse suivante considère un ordre $N = 2$ à 10.

Les formes directes 1 et 2 possèdent $2N + 1$ opérations de multiplication. Il en vient que le nombre d'architectures qui correspondent à ces deux formes est :

$$2 \times \sum_{N=2}^{10} 2N + 1 = 234$$

Les équations d'état donnent $(N + 1)^2$ opérations de multiplication. Cela donne le nombre d'architectures possibles :

$$\sum_{N=2}^{10} (N + 1)^2 = 501$$

Pour les structures décomposées, le nombre de sections qui doivent être utilisées pour un filtre d'ordre N est la valeur entière de : $log_2 N$. Il y a donc 29 architectures en sections pour les ordres $N = 2$ à 10 et pour la structure cascade. Le même nombre existe pour la structure

parallèle ce qui donne 58 architectures. Pour une section d'ordre $N = 2$, il y a encore $2(2N + 1) = 10$ architectures pour les formes directes 1 et 2 et $(N + 1)^2 = 9$ architectures à partir des équations d'état. Le nombre total des architectures décomposées sera donc : $(9 + 10) \times 58 = 1102$ architectures. En total, il ya 234+501+1102=1837 architectures. C'est le nombre théorique des architectures. Beaucoup de considérations pratiques peuvent diminuer ce nombre jusqu'à la moitié.

Les éléments suivants caractérisent chaque architecture dans l'espace des solutions :

N : l'ordre du filtre ;

F_e : la fréquence d'échantillonnage ;

B : la largeur en bits des multiplieurs ;

V_M : le nombre de pas de contrôle dédiés au test en-ligne pendant une période d'échantillonnage ;

N_{Mult} : le nombre de multiplieurs au niveau RTL ;

S : la structure de réalisation ;

M_{test} : la méthode de test en-ligne ;

N_{ctrl} : le nombre de pas de contrôle de la période fonctionnelle ;

F_{Op} : la fréquence opérationnelle du filtre.

L'ordre du filtre, la fréquence d'échantillonnage et la largeur en bits des opérateurs sont liés directement au fonctionnement et à la stabilité du filtre et sont considérés fixés par l'utilisateurs. L'outils peut proposer de changer l'un de ces éléments si les contraints imposées au système ne sont pas satisfaites.

Pour le test en-ligne non-concurrent, le nombre de vecteurs de test pour un multiplieur dépend de sa largeur en bits B. Cette valeur est recherché dans une base de données qui contient cette information pour chaque largeur en bit d'unités fonctionnelles (figure (6.7)). V_M est calculé à partir de cette valeur et de la valeur de la latence de faute imposée sous forme de contrainte de test en-ligne.

Côté test en-ligne semi-concurrent, V_M représente le nombre de pas de contrôle nécessaires pour la vérification des unités fonctionnelles (exploitation de la distributivité) ou pour compléter l'ordonnancement du graphe de test en-ligne (vérification globale du calcul).

Le nombre de pas de contrôle ainsi que la fréquence opérationnelle du filtre sont calculés à partir des valeurs de N_{Mult}, S et M_{test}.

6.7.1 Chromosome

Les éléments qui caractérisent chaque architecture dans l'espace des solution, voir la section précédente, sont utilisés pour représenter un individu dans une population. Chaque individu aura la forme suivante :

N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}

Trois parties peuvent être distinguées dans ce chromosome. Ce sont la prtie obligatoire, la partie facultative et la partie calculée.

La première partie, celle obligatoir, se compose de trois éléments : l'ordre du filtre N, la fréquence d'échantillonnage F_e et la largeur en bit des opérateurs B. Ces valeurs concernent directement le calcul des coefficients du filtre et ses caractéristiques fonctionnelles. Notre programme de compilation et d'ordonnancement ne se permet de changer ces valeur que sous forme de propositions si aucune autre solution n'est pas trouvée.

La deuxième partie est celle facultative. Cette partie conserne le nombre de multiplieurs au niveau RTL N_{Mult}, la structure de réalisation S et la méthode de test en-ligne M_{test}. C'est la partie qui va servir essentiellement pour l'exploration de l'espace de solutions. Le fait que ces éléments soient facultatifs est limité par le fait que le concepteur peut les imposer sous forme de contraintes.

La troixième partie est calculée à partir des éléments précédents. Elle se compose du nombre de pas de contrôle du graphe de flot de données ordonnancé N_{ctrl}, du nombre de pas de contrôle dédiés au test en-ligne V_M et de la valeur de la fréquence opérationnelle du système F_{Op}.

Il y a deux approches pour le calcule de la fréquence opérationnelle. La première consiste à calculer la fréquence opérationnelle maximale que permet le cible technologique. Pour ce faire, le délai du multiplieur est utilisé à partir de la base de données qui contient les caractéristiques des unités fonctionnelles. A partir de la valeur obtenue pour la fréquence opérationnelle, le nombre de pas de contrôle de l'intervalle oisif de la période d'échantillonnage peuvt être calculé. Ce nombre doit être suppérieur ou égale à V_M calculé à partir de la valeur de la latence de faute imposée au système. La deuxième approche pour le calcul de la fréquenc opérationnelle consiste à déterminer d'abord la structure de réalisation et le nombre de multiplieurs pour définir l'architecture et calculer le nombre total de pas de contrôle de l'intervalle fonctionnelle N_{ctrl}. Ensuite, V_M est calculé à partir de la latence de faute imposée. Enfin, la fréquence opérationnelle nécessaire pour faire fonctionner le système est calculée. L'avantage de la deuxième approche est de faire fonctionner le système à la fréquence opérationnelle minimale pour minimiser la consommation du système.

6.7.2 Codage

Un codage mixte du chromosome est choisi. Une partie des éléments construisant le chromosome est représentée en décimal alors que l'autre partie est représentée en binaire.

Les éléments codés en décimal sont ceux de la première et de la troixième partie (obligatoire et calculée). Les éléments de la partie facultative sont codés en binaire. La méthode de test en-ligne M_{test} prend deux valeurs, NC= non-concurrent et SC= semi-concurrent avec un bit de code associé 0 et 1 respectivement. Le nombre de multiplieurs au niveau RTL est codé sur 7 bits pour pouvoir manipuler des filtre d'ordre $N > 7$ décrit en équations d'état. La structure de réalisation est codée sur deux bits. Elle prennent les valeurs indiquées dans le tableau (6.1).

6.7.3 Fonction d'évaluation

L'évaluation de chaque individu se fait par le contrôle de trois éléments : V_M, N_{Mult} et N_{ctrl}. Ces éléments représentent respectivement les contraintes de test en-ligne, les contraintes de surface et les contraintes de délai. La distance entre les valeurs réferences calculées à partir

Structure	Directe 1	Directe 2	Equations d'état	Décomposée
S	D1	D2	EE	DEC
Code	00	01	10	11

Table 6.1: *Codage de la structure de réalisation.*

des contraintes imposées et les valeurs de ces éléments pour chaque individu détermine sa "*fitness*". Les définitions suivantes vont servire au calcul de la "*fitness*" :

$L_{VM} = V_{M_{ref}} - V_M$: la différence entre la valeur réference imposée par les contraintes de test en-ligne et V_M ;

$L_{Mult} = N_{Mult_{ref}} - N_{Mult}$: la différence entre la valeur réference imposée par les contraintes de surface et N_{Mult} ;

$L_{ctrl} = N_{ctrl_{ref}} - N_{ctrl}$: la différence entre la valeur réference imposée par les contraintes de délai et N_{ctrl}.

f_{VM} : la "*fitness*" en testabilité en-ligne ;

f_{Mult} : la "*fitness*" en surface ;

f_{ctrl} : la "*fitness*" en délai.

La "*fitness*" de l'individu comporte donc trois valeurs. Ces valeurs sont calculées par les relations suivantes :

$$f_{VM} = \begin{cases} \frac{1}{|L_{VM}|+1} & ; L_{VM} > 0 \\ 1 & ; L_{VM} = 0 \\ L_{VM} & ; L_{VM} < 0 \end{cases}$$

$$f_{Mult} = \begin{cases} \frac{1}{|L_{Mult}|+1} & ; L_{Mult} < 0 \\ 1 & ; L_{Mult} = 0 \\ L_{Mult} & ; L_{Mult} > 0 \end{cases}$$

$$f_{ctrl} = \begin{cases} \frac{1}{|L_{ctrl}|+1} & ; L_{ctrl} < 0 \\ 1 & ; L_{ctrl} = 0 \\ L_{ctrl} & ; L_{ctrl} > 0 \end{cases}$$

Le résultat suivant peut être tiré de ces fonctions d'évaluation :

$1 > fitness > 0$: l'individu ne satisfait pas aux contraintes imposées ;

$fitness = 1$: l'individu représente exactement les contraintes imposées ;

$fitness > 1$: l'individu satisfait les contraintes imposées avec une marge de plus.

6.8 Reproduction

L'exploration de l'espace de solution se fait sous forme d'une reproduction répétée des individus et une évaluation de la nouvelle population. La reproduction commence par la génération d'une population initiale. Les individus dans cette population sont évalués et certains[1] sont sélectionnés pour la reproduction. Les opérateurs génétiques sont appliqués aux individus sélectionnés. Les progénitures sont insérées dans la population des parents pour générer une nouvelle population.

6.8.1 Population initiale

La population initiale peut se composée d'individus sélectionnés aléatoirement dans l'espace de solutions. Cependant, une bonne sélection de cette population peut aider de façon importante à une convergence rapide de l'algorithme vers la solution souhaitée. Nous utilisons une heuristique permettant une génération déterministe de la population initiale.

Pour les filtres numériques, l'espace de solutions comporte plusieurs structures de réalisation avec plusieurs architectures possibles pour chaque structure, voir la section 6.7. La génération de la population initiale se base sur la sélection des individus qui représentent les différentes structures de réalisation et qui satisfaisent au moins à une des contraintes imposées.

Dans un premier temps, les individus qui satisfaisent à la latence de faute sont générés. Pour chaque structure, les architectures parallèle, série et éventuellement intermédiaire qui représentent une latence de faute égale à 1 sont sélectionnées. La valeur de V_M est divisée par la latence de faute imposée par les contraintes de test en-ligne et exprimée en nombre de périodes d'échantillonnage. A rappeler qu'une latence de faute égale à 1 signifie qu'un test complet[2] est appliqué au système dans une période d'échantillonnage. Ces différentes architectures ont été explorées dans les sections 6.4 et 6.5.

Dans un deuxième temps, les individus qui satisfaisent à la fois aux contraintes de test en-ligne et aux contraintes de surface sont générés à partir des premiers individus. D'abord, un nouveau individu qui hérite N, F_e, B, V_M, S et M_{test} est inséré pour chaque structure de réalisation. Ensuite, le nombre de multiplieurs imposé par les contraintes de surface est associé à N_{Mult} pour chaque nouveau individu. Enfin, N_{ctrl} est évalué et utilisé avec V_M pour calculer la fréquence opérationnelle F_{Op} par la relation :

$$F_{Op} = F_e(N_{ctrl} + V_M)$$

Cette génération de la population initiale est illustré par l'exemple suivant.

Considérons la fonction de transfert suivante qui représente un filtre IIR d'ordre $N = 3$:

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}{1 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}$$

Cette fonction de transfert contient $2N + 1 = 7$ opérations d'opérations de multiplication. L'architecture série et celle parallèle qui correspondent à sa rélisation par la structure forme directe 1 sont données par la figure (6.20).

Supposons que ce filtre est conçu pour fonctionner à une fréquence d'échantillonnage $F_e = 100KHz$ et que des multiplieurs 16 bits sont utilisés au niveau RTL. Supposons encore

[1] Ou tous.

[2] Qu'il soit structurel (test non-concurrent) ou fonctionnel (test semi-concurrent).

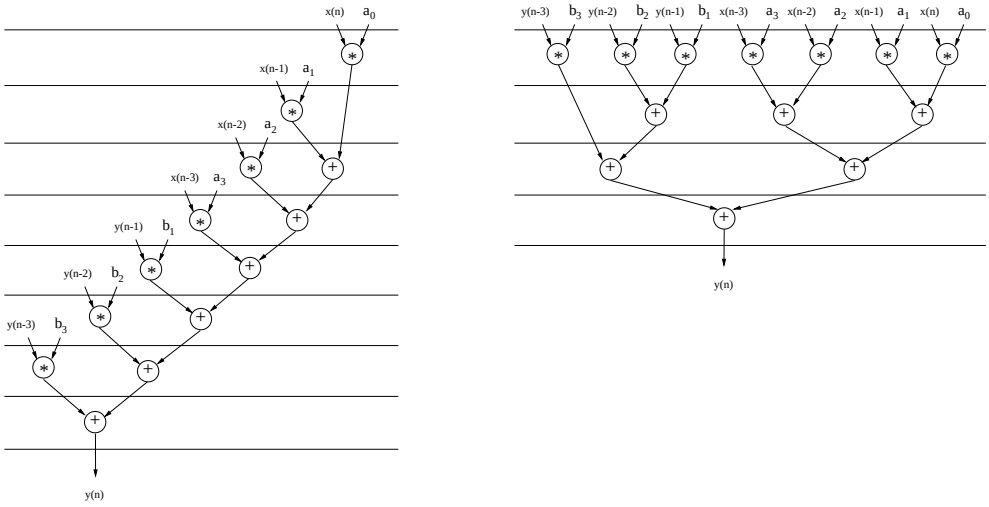


Figure 6.20: *La génération de la population initiale pour un filtre IIR N= 3 et la structure forme directe 1 : l'architecture série (a) et celle parallèle (b) pour .*

que la latence de faute est fixée à 15 μs, c'est à dire, 1.5 de périodes d'échantionnage et que les contraintes de surface limitent le nombre de multiplieurs à 2 multiplieurs.

Les individus qui satisfaisent à la latence de faute et qui représentent les différentes structures de réalisation sont générés. Pour la méthode de test en-ligne non-concurrent, si l'on considère la technomogie 0.6μ cub de AMS comme cible technologique, il y a 59 vecteurs de test à appliquer sur les multiplieurs, voir la figure (6.7). Pour obtenir une latence de faute égale à 1, tous les vecteurs de test doivent être appliqués aux miltiplieurs pendant une période d'échantillonnage. La latence de faute imposée au système est de 1.5 de périodes d'échantillonnage. Le nombre devecteurs de test à appliquer dans une période d'échantillonnage devient donc: $\frac{59}{1.5} = 39.33$ ou 40 vecteurs de test. Pour satisfaire à cette valeur il faut que le filtre puisse fonctionner à une fréquence opérationnelle :

$$F_{Op} = F_e(N_{ctrl} + V_M) = 100KHz \times (8 + 40) = 4.8MHz \text{ pour l'architecture série}$$

et

$$F_{Op} = F_e(N_{ctrl} + V_M) = 100KHz \times (4 + 40) = 4.4MHz \text{ pour l'architecture parallèle}$$

Les chromosomes qui représentent ces deux architectures sont donc :

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}
Forme directe 1	Architecture série	3	0.1	16	40	1	D1	NC	8	4.8
	Architecture parallèle	3	0.1	16	40	7	D1	NC	4	4.4

De la même façon, les individus qui représentent les architectures série et parallèle pour la structure forme directe 2 et les équations d'état sont générés. Ces individus correspondent aux chromosomes suivants :

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}
Forme directe 2	Architecture série	3	0.1	16	40	1	D2	NC	8	4.8
	Architecture parallèle	3	0.1	16	40	7	D2	NC	3	4.3
Equation d'état	Architecture série	3	0.1	16	40	1	EE	NC	17	5.7
	Architecture parallèle	3	0.1	16	40	16	EE	NC	5	4.5
	Architecture intermédiaire	3	0.1	16	40	4	EE	NC	6	4.6

A partir des individus générés, un nouveau individu est inséré pour chaque structure de réalisation. Ce nouveau individu hérite les parties communes entre les individus de la même structure de réalisation. $N_{Mult} = 2$ est associé à chaque nouveau individu et N_{ctrl} et F_{Op} sont calculés. La population initial est donc modifiée est devient:

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}
Forme directe 1	Architecture série	3	0.1	16	40	1	D1	NC	8	4.8
	Architecture parallèle	3	0.1	16	40	7	D1	NC	4	4.4
	Nouvelle architecture	3	0.1	16	40	2	D1	NC	6	4.6
Forme directe 2	Architecture série	3	0.1	16	40	1	D2	NC	8	4.8
	Architecture parallèle	3	0.1	16	40	7	D2	NC	3	4.3
	Nouvelle architecture	3	0.1	16	40	2	D2	NC	5	4.5
Equation d'état	Architecture série	3	0.1	16	40	1	EE	NC	17	5.7
	Architecture parallèle	3	0.1	16	40	16	EE	NC	5	4.5
	Architecture intermédiaire	3	0.1	16	40	4	EE	NC	6	4.6
	Nouvelle architecture	3	0.1	16	40	2	EE	NC	10	5.0

Les individus qui représentent les architectures réalisées avec la méthode de test en-ligne semi-concurrent sont aussi générés. Une représentation graphique de cette génération est dans l'intersection d'une colonne $(F_e, N) = (100KHz, 3)$ avec les différentes surfaces qui

représentent une latence de faute égale à 1, voir figures(6.17 à 6.19). Cette solution est présentée dans la figure (6.21)

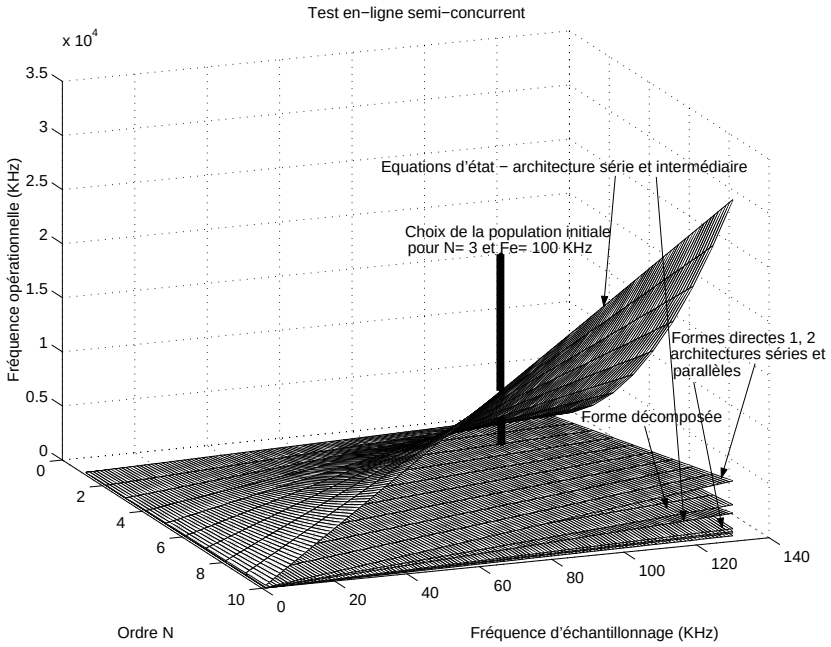


Figure 6.21: *Choix de la population initiale : test semi-concurrent.*

6.8.2 Sélection

Cet opérateur a comme but de déterminer les parents pour une nouvelle génération. Deux chromosomes sont sélectionnés pour la génération de deux progénitures. Pour la population initiale, un individus représentant une architecture série est sélectionné avec un autre qui représente une architecture parallèle et éventuellement intermédiaire.

Avec cette sélection, l'individu qui représente l'architecture série de la forme directe 1 est sélectionné avec celui qui représente l'architecture parallèle de la même structure. Deux progénitures sont générées de ces deux parents. La même opération est réalisée pour la forme directe 2 et deux autres progénitures sont ajoutées à la population initiale. Pour les équations d'état, l'individu qui représente l'architecture série est sélectionné deux fois ; pour les individus qui représentent les architectures parallèle et intérmidiaire respectivement. Quatre progénitures sont générées de cette opération.

La taille maximale de la population est limitée à 30 individus ; 15 individus pour le test en-ligne non-concurrent et le même nombre pour le semi-concurrent. Cette taille peut être diminuée pour des ordres moins important afin de diminuer la complexité de l'algorithme de l'exploration.

Dans une nouvelle génération, quand le nombre d'individus dépasse la limite, les individus ayant la meilleurs valeur de "*fitness*" sont gardés et les autres sont éliminés de la population. La sélection des parents est dons réalisée en utilisant le principe de roulette de Wheel. Pour

cela, la somme de "*fitness*" de tous les individus de la population est calculée et un nombre aléatoire entre 0 et cette somme est généré. Le premier individu dont la somme de "*fitness*" avec celle de l'individu précédent dépasse ce nombre est sélectionné. Cette sélection est répétée pour choisir les parents de la nouvelle génération jusqu'à ce que tous les individus soient sélectionnés. Cette opération se fait sans remplacement, c'est à dire, les individus ne sont pas reinjectés dans la population après leur sélection.

6.8.3 Crossover

Cet opérateur est essentiel dans le processu de la génération de nouveau individus. Il consiste à croiser les gènes de deux chromosomes pour produire deux nouveaux gènes qui seront utilisés pour former deux nouveaux chromosomes. L'exemple dans la figure (6.22) montre cette opération pour deux individus sélectionnés dans la population initiale de l'exemple du filtre IIR 3ème ordre de la section précédente.

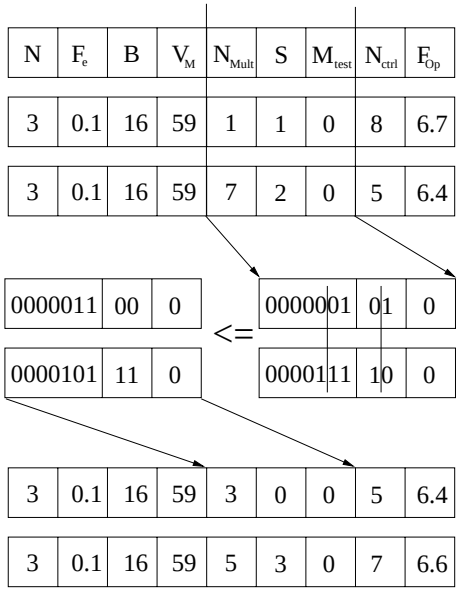


Figure 6.22: *Opérateur : "crossover".*

6.8.4 Mutation

La mutation est un opérateur génétique qui permet de brasser la population par la génération d'individus qui ne peuvent pas être générés directement par un simple "*crossover*". Dans un algorithme génétique classique cet opérateur est appliqué aléatoirment avec une très faible propabilité, de l'ordre de 0.1%. Dans notre cas, la mutation sera utilisée à deux stage de la génération. La première utilisation est déstinée à modifier les solutions ayant $N_{Mult} = 0$. Ce cas n'existe pas en pratique et à la place d'éliminer une telle solution elle sera modifiée pour générer une solution réalisable. La mutation dans ce cas est objective. Sa probabilité est égale à la propabilité d'obtenir $N_{Mult} = 0$ dans une génération. La deuxième utilisation

est clasique et s'applique à N_{Mult} et à la méthode de test en-ligne M_{test} qui est codée sur un seul bit et ne peut pas subir une opération de "*crossover*".

Cete deuxième application de la mutaion est propabilistique. Nous considérons une propabilité de mutation égale à 0.1% pour chaque bit de N_{Mult} et pour le bit de M_{test}. Pour cela, deux nombres aléatoires sont générés. Le premier est entre 1 et 10000 et servira à déterminer si la mutation aura lieu ou non. Le deuxième est entre 0 et 8 et détermine le bit qui va subir cette mutation. Si le premier nombre est inférieur à 10 la mutation est réalisée sur le bit qui correspond au deuxième nombre. Le nombre 0 est associé à M_{test} alors que les nombres de 1 à 8 sont associés à N_{Mult}.

6.8.5 Nouvelle génération

Après l'application des opérateurs génétiques, la nouvelle population se trove composée des parents et des progénitures. En raison de la limitation de la taille de la population à 30 individus, un tri est réalisé pour sélectionner les 30 meilleurs individus et éliminer le reste de la population. Dans un premier temps, tous les individus dupliqués sont éliminés. Si le nombre d'individus dans la population reste élevé, un deuxième tri se fait lors de l'évaluation de la "*fitness*" des individus pour décider si un individu doit être retenu ou éliminé.

6.9 Algorithme de l'exploration

La géneration de l'architecture RTL testable en-ligne à partir des spécifications comportementales du système se fait par une exploration de l'espace de solutions. L'algorithme dans la figure (6.23) montre les différentes étapes de cette exploration.

Dans un premier temps, une analyse de ces spécifications permet l'établissement des parties obligatoires du chromosome. Une première base de données contenant des expériences intérieures est consultée pour déterminer une ou plusieurs des parties facultatives. Essentiellement cela servira à la détermination de la structure de réalisation et de la méthode de test en-ligne.

Cette base de données est établie au fur et à mesure que l'algorithme traite de nouveaux systèmes. Une nouvelle expérience (régle) est introduite dans cette base quand l'algorithme traite de nouvelles spécifications comportementales d'un système et lui trouve une solution. Cette solution est donc attribuée à ces spécifications comportementales avec une propabilité de 1%. Si les même spécifications sont utilisées une deuxième fois pour décrire un système et que la même solution est trouvée le pourcentage d'attribution de cette solution à ces spécifications passe de 1% à 2%. Quand cette propabilité passe un seuil de 20% la solution est inséreée automatiquement dans la population initiale. Pour une propabilité égale ou supérieure à 50% la solution est évaluée avant la génération de la population initiale.

Une deuxième base de données contient les caracteristiques des unités fonctionnelles utilisées au niveau RTL selon le cible technologique. Ces caracteristiques concernent la surface des differentes unités fonctionnelles pour les differentes largeurs en bits, le délai de ces unités et un ensemble de vecteurs de test garantissant une couverture élevée de faute pour chaque type d'unités fonctionnelles et chaque largeur en bits.

Pour le test en-ligne non-concurrent, la deuxième base de données est consultée pour obtenir le nombre de vecteurs de test pour les unités fonctionnelles du chemin de données. Ce nombre de vecteurs de test est utilisé avec la latence de faute imposée par les contraintes de test en-ligne et avec le nombre des temps oisifs dans l'intervalle fonctionnel pour calculer

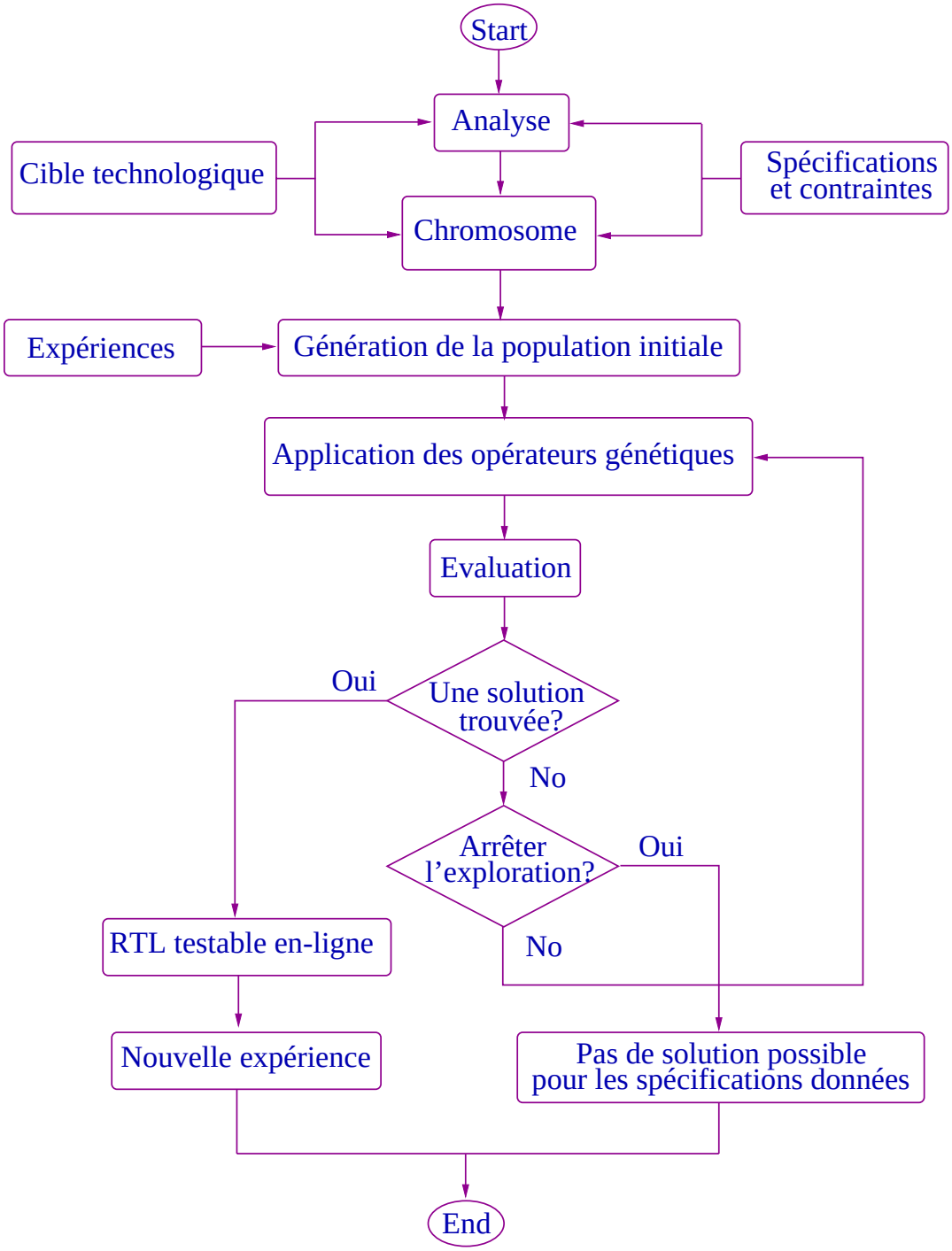


Figure 6.23: *Exploration de l'espace de solutions par l'algorithme génétique.*

V_M. Par exemple, pour un multiplieur Mult, soit N_{Mult} le nombre de vecteurs de test. Soit I_mult_f le nombre de temps oisifs du multiplieur dans l'intervalle fonctionnel. Si les contraintes de test en-ligne imposent une latence de faute L_{faute} périodes d'échantillonnage, c'est à dire, l'ensemble de vecteurs de test doit être appliqué $\frac{1}{L_{faute}}$ fois dans une période d'échantillonnage, dans ce cas:

$$V_M = \frac{V_{Mult}}{L_{faute}} - I\_mult_f$$

Pour le test en-ligne semi-concurrent, V_M exprime le nombre supplémentaire de pas de contrôle nécessaire pour achever l'ordonnancemnt du graphe de test en-ligne. La même relation qui a été utilisée pour calculer V_M pour le test en-ligne non-concurrent peut être utilisée pour le test en-ligne semi-concurrent. Il faut seulement remplacer V_{Mult} par le nombre total des pas de contrôle du graphe de test en-ligne et remplacer I_{Mult} par le nombre de pas de contrôle du graphe de test en-ligne qui sont ordaonnacés dans l'intervalle fonctionnel.

En général, V_M représente le nombre de pas de contrôle qui doit être disponible dans l'intervalle oisif pour terminer le test en-ligne qui a été commencé dans l'intervalle fonctionnel.

L'exploration continue tant qu'une solution n'est pas trouvée ou une limite en temps d'exécution ou en nombre de générations n'est pas dépassée. La limite imposée dans notre cas dépend du système synthétisé. Pour être raisonnable en temps d'exécution de l'algorithme, cette limite doit avoir une relation linéaire avec la complexité du système traité. Cela permet de ne pas faire ni une exploration excessive pour un système simple ou une exploration exhaustive serait plus efficace, ni une exploration insuffisante pour un système complex. Nous considérons la relation suivante pour le calcul de cette limite en temps d'exécution:

$$G \times I < A$$

ou G est le nombre de génération, I est le nombre d'individus dans la population et A est le nombre d'architectures dans l'espace de solution du système synthétisé.

6.10 Conclusion

Dans cette partie, le problème de la synthèse de haut niveau pour le test en-ligne a été abordé. Des solutions à plusieurs niveaux ont été proposées.

Premièrement, une optimisation, orientée testabilité en-ligne, de la description comportementale a été proposée. Cette optimisation peut apporter un gain considérable en testabilité en-ligne (chapitre 5).

Ensuite, une méthode de compilation et d'ordonnancement a été élaborée. Cette méthode permet, d'une part, la prise en compte des contraintes de test en-ligne, de surface et de délai et permet, d'une autre part, la manipulation de systèmes complexes. Un algorithme génétique a été combiné avec une heuristique pour l'exploration de l'espace de solutions. Les filtres numériques récursifs ont été adressés par cette étude. Les différentes structures de réalisation et les différentes architectures ont été analysées.

Enfin, une méthode de test en-ligne peut être sélectionnée et intégrée au système au niveau ordonnancement. A noter que ces différentes étapes peuvent fonctionner de façon indépendante, c'est à dire, l'algorithme qui permet la génération de l'architecture RTL

testable en-ligne peut avoir comme entrée une description comportementale, un graphe de flot de données non-ordonnancé ou un graphe de flot de données ordonnancé.

La figure (6.24) montre l'intégration de ces solutions dans le flot général de HLS_OLT proposé par cette étude.

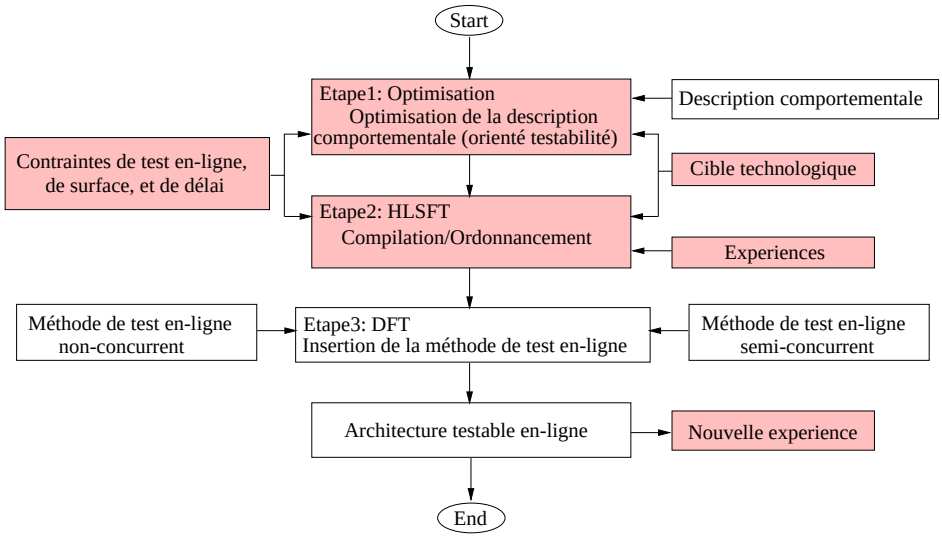


Figure 6.24: *L'intégration des méthodes d'optimisation et de compilation et ordonnancement dans le flot de la synthèse de haut niveau pour le test en-ligne (HLS_OLT).*

La dernière partie de cette étude comprend deux chapitres. Le premier chapitre présente un flot général qui regroupe les différentes méthodes développées dans les chapitres précédents. Le deuxième chapitre présente une application sur un filtre numérique récursif du quatrième ordre. Les différentes étapes de la méthode de synthèse de haut niveau pour le test en-ligne qui ont été développées tout au long de cette étude sont illustrées par cette application.

Partie IV

Solution intégrée pour HLS_OLT

Chapitre 7

Algorithme général

Les différentes méthodes étudiées dans les chapitres précédents permettent la définition d'un nouveau flot de synthèse de haut niveau pour le test en-ligne. Les contraintes de test en-ligne ainsi que celles de surface et de délai sont prises en compte dans ce nouveau flot. Les étapes de ce flot de synthèse seront résumées dans la suite de ce chapitre et un algorithme général sera présenté. Il est util de rappler que ce nouveau flot peut être introduit à plusieurs niveaux dans la synthèse de haut niveau. Il est possible de commencer par une description comportementale, faire éventuellement une optimisation orientée test en-ligne, générer le graphe de flot de données ordonnancé par compilation et ordonnancement et enfin intégrer une méthode de test en-ligne au graphe de flot de données ordonnancé. Il est également possible de partir d'un graphe de flot de données ordonnancé ou non-ordonnancé pour générer l'architecture testable en-ligne. Cela permet de définir ces étapes de façon indépendante, c'est à dire, chaque étape doit vérifier la conformité de la solution aux contraintes imposées.

7.1 Flot de synthèse de haut niveau pour le test en-ligne

Le système, donné sous forme de spécifications de paramètres ou sous forme d'une description VHDL comportementale, voir annexe A, est analysé pour adopter une méthode de test en-ligne. Plusieures méthodes de test en-ligne non-concurrent et semi-concurrent sont proposées. Cette analyse concerne trois aspects: les spécifications fonctionnelles du système, les contraintes imposées et le cible technologique de l'implémentation. Des expériences antérieures peuvent être utilisées pour la prise de décision lors de cette analyse.

La factorisation des équations arithmétiques permet à l'étape de la compilation de générer un graphe de flot de données optimisé pour le test en-ligne. Elle permet aussi la génération des variante-duales utiles pour l'implémentation de la méthode de test en-lign semi-concurrent.

La compilation des équations optimisées en graphe topologique peut générer un ou plusieurs graphes de flot de données ordonnancés qui satisfaisent aux contraintes de test en-ligne, de surface et de délai. Un de ces graphes est choisi pour l'implémentation du système alors que les autres graphes servent comme des variante-duales et peuvent être utilisés comme graphe de test en-ligne pour le test semi-concurrent.

La méthode de test en-ligne adoptée est intégrée au système au niveau ordonnancement. L'ordonnancement nominal est modifié par l'insertion des opérations de test en-ligne dans les temps oisifs des unités fonctionnelles. Une allocation de ressources et une assignation permettent la génération de l'architecture testable en-ligne du système.

Optimisation de la description comportementale

Cette optimisation s'applique à une description comportementale contenant des équations arithmétiques. Elle est basée sur la factorisation de ces équations de façon à améliorer la testabilité en-ligne du design final. Les équations sont analysées pour identifier les meilleures variables communes pour la testabilité en-ligne.

Dans un premier temps, tous les variables utilsés dans ces équations sont identifiés et la matrice de corrélation de variables est établie. Ensuite, les variables qui représentent le meilleurs gain sont sélectionnés comme les meilleures variables communes. Enfin, la factorisation est réalisée par les variables sélectionnés.

Cette optimisation est orientée testabilité en-ligne par le fait qu'elle favorise l'existence des temps oisifs dans le graphe de flot de données ordonnancé. Ainsi la testabilité en-ligne est améliorée aussi bien pour les méthodes de test en-ligne non-concurrent que pour les méthodes de test en-ligne semi-concurrents. En effet, l'augmentation des temps de repos facilite l'application des vecteurs de test pour le test en-ligne non-concurrent et facilite l'insertion du graphe de test en-ligne pour le test en-ligne semi-concurrent.

Compilation et ordonnancement du graphe de flot de données

La compilation et l'ordonnancement sont formés comme un seul problème d'exploration d'espace de solutions. La solution à ce problème est obtenue par l'utilisation d'un algorithme génétique. Le résultat est un ou plusieurs graphes de flot de données ordonnancés en tenant compte des contraintes de test en-ligne, de surface et de délai.

Deux bases de données sont consultées lors de cette exploration. La première contient des expériences antérieures qui peuvent guider l'exploration pour de nouvaux systèmes. La deuxième contient les caracteristiques des unités fonctionnelles à utiliser dans le chemin de données. Ces caracteristiques sont utilisées pour la construction du chromosome.

La limitation du nombre de génération dépend de la complexité du système synthétisé et augmente linéairement avec cette complexité.

pour des considérations pratiques et afin d'améliorer les performances de l'algorithme, une heuristique est combinée avec l'algorithme génétique. Cette heuristique est introduite à l'étape de la génération de la population initiale. Elle consiste à générer plusieurs individus qui représentent les différentes architectures disponibles dans l'espace de solutions. En plus, les contraintes de test en-ligne et celles de surface sont considérées lors de cette génération. Cette combinaison permet, d'un côté, de profiter de l'approche génétique pour faciliter l'exploration heuristique pour de problèmes simples ou à complexité moyenne et, d'un autre côté, de pouvoir manipuler de problèmes très complexes à un coût raisonnable en temps d'exécution et en ressource.

Analyse du graphe ordonnancé et choix de la méthode de test en-ligne

Le choix de la méthode de test en-ligne dépend de la disponibilité des temps oisifs dans le graphe de flot de données ordonnancé. Les systèmes qui disposent d'un intervalle oisif important peuvent être implémentés avec un circuit de test en-ligne non-concurrent, alors que les systèmes qui ne disposent pas d'un tel intervalle ou pour lesquels cet intervalle est insuffisant sont implémentés avec le test en-ligne semi-concurrent.

D'abord, la différence entre l'intervalle fonctionnel et la période d'échantillonnage est calculée. La latence de faute qui peut être ainsi obtenue est évaluée. Si sa valeur ne satisfait pas aux contraintes de test en-ligne, les temps oisifs dans l'intervalle fonctionnel sont identifiés et la latence de faute totale de la période d'échantillonnage est calculé. Si les contraintes de test en-ligne ne sont toujours pas satisfaites, les méthodes du test en-ligne semi-concurrent sont explorées.

Pour les méthodes de test en-ligne semi-concurrent, le graphe nominal est utilisé comme un graphe de test en-ligne. Si ce graphe ne permet pas d'atteindre la latence de faute demandée, les variante-duales disponibles du graphe nominal sont explorées.

Si ni les méthodes de test en-ligne non-concurrent ni les méthodes de test en-ligne semi-concurrent ne permettent pas de satisfaire aux contraintes imposées, les méthodes de test en-ligne concurrent sont envisagées. L'algorithme en figure (7.1) implémente ce nouveau flot de synthèse de haut niveau pour le test en-ligne.

7.2 Domaine d'application

Le flot de la synthèse de haut niveau pour le test en-ligne qui vient d'être défini sera appliqué aux filtres numériques. Plus particulièrement, les filtres elliptiques récursifs sont adressés. Le choix de ce type de systèmes numériques peut être justifié par plusieures raisons.

D'abord, les systèmes de traitment numérique de signal représentent un catégorie important de systèmes numériques dans plusieurs domaines pratiques ; citons par exemple les applications médicales, spatiales et celles de la communication et du contrôle.

Ensuite, les filtres numériques récursifs représentent un cas général de filtres numériques et souffrent de problèmes d'instabilité. Il en vient qu'une application réussite à ce type de systèmes numériques peut être très facilment généralisée pour les filtres non-récursifs (FIR). En effet, l'algorithme génétique de la compilation et ordonnancement offre une solution convenable à la complexité des filtres numériques ayant un ordre N élevé. Par ailleurs, les circuits de test en-ligne sont complètemet transparents par rapport au type du filtre.

Finalement, les structures de réalisation ont été explorées et analysées pour le test en-ligne non-concurrent et semi-concurrent. Ces structures sont la forme directe 1, la forme directe 2, les équations d'état et les formes décomposées. Ces formes servent aussi bien à la génération des filtres récursifs (IIR) qu'à la génération des filtres non-récursifs (FIR).

7.3 L'outil : HLS_OLT

L'outil conçu pour démontrer la solution proposée de synthèse de haut niveau pour le test en-ligne se compose de plusieurs programmes en C++ et de plusieurs scripts permettant la réalisation des différentes tâches. Pour faciliter l'utilisation de cet outil une interface graphique (*GUI*) a été conçu en Tcl/Tk.

D'abord, le fait de concevoir une telle *GUI* permet une utilisation simplifiée et consistante de cet outil. Elle permet aussi de regrouper les clés de contrôle des différentes tâches ainsi que les différents messages communiqués par ces tâches dans un seul endroit.

Ensuite, le choix de Tcl/Tk pour réaliser cet interface graphique est justifié par plusieures raisons. Ces raisons sont résumées dans les points suivants [1] :

- Tcl (*Tool Command Language*) est employé par plus d'un demi-million de réalisateurs dans le monde entier et est devenu un composant critique dans des milliers de sociétés.

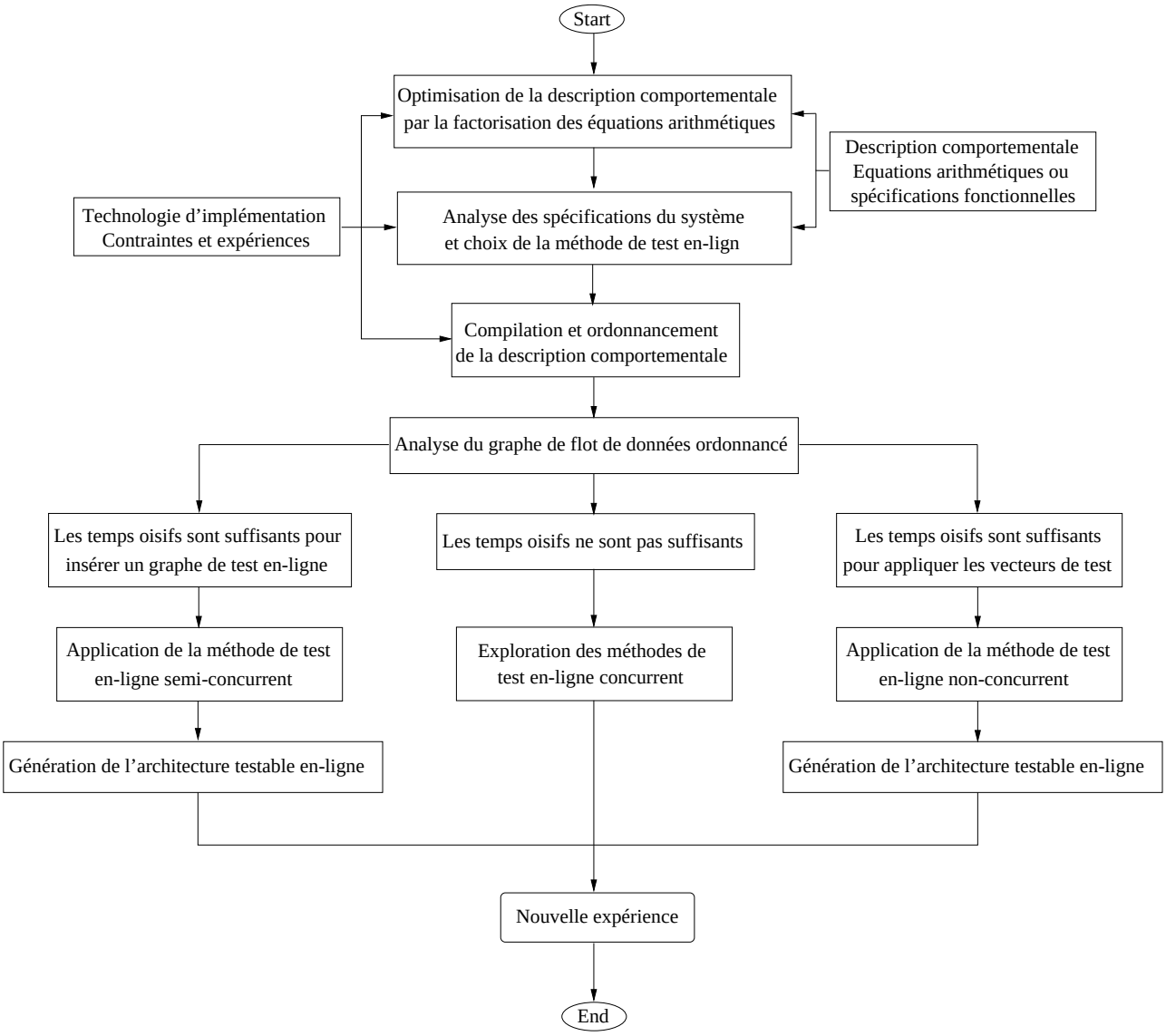


Figure 7.1: *Le flot de la méthode de synthèse de haut niveau pour le test en-ligne.*

Il a une syntaxe simple et programmable et peut être employé comme application autonome ou être enfoncé dans des programmes d'application. En plus, le TCL est une source ouverte (*open source*) ainsi il est complètement gratuit ;

- Le Tk est un toolkit d'interface graphique pour l'utilisateur. Il permet, très rapidement, la création d'interfaces graphiques (*GUIs*) puissants. Sa large popularité est prouvée par le fait qu'il soit avec toutes les distributions de Tcl ;

- Les programmes et scripts Tcl/Tk sont largement portables. Ils peuvent tourner sur toutes les architectures Unix telles que Linix, Solaris, IRIX, AIX, *BSD* ... ainsi que Windows, Macintosh et encore d'autres. Il y a plusieures sites internet qui proposent des sources binaires précompilées pour des différentes platforms ;

- L'utilisation de Tcl/Tk est très répandu dans le monde de "*Electronic Design Automation*" (EDA) et de "*Computer-Aided Design*" (CAD). Citons par exemple Cadence AmBit (Tcl/Tk), Cadence SignalScan (Tk), Cadence NCSim (Tcl), ModelSim (Tcl/Tk) et Synplcity Synplify (Tcl/MFC) ;

- Le Tcl/Tk permet la réalisation des applications extensibles, une intégration fléxible de nouveaux composants dans l'application et est facile à apprendre.

L'interface graphique conçu pour l'outil en Tcl/Tk ainsi que ses menus sont présentés en figure (7.2). L'objet de chaque menu sera exposé dans la suite de cette section. Le script Tcl/Tk est présenté en annexe E.

- **File :**

  Open ... : permet au utilisateur d'ouvrir un fichier quelconque ;

  Log file : permet la visualisation du fichier d'information de l'état de l'outil ;

  VHDL filter : permet l'ouverture du fichier VHDL contenant la description RTL testable en-ligne du filter spécifié ;

  Target library script : permet l'ouverture du fichier script utilisé pour la génération de la base de données de la technologie cible, voir annexe E ;

  Quit : permet de quiter l'outilen nettoyant le répertoire du travail d'éventuels fichiers temporairs.

- **Set :**

  Filter specifications : ouvre une fenêtre permettant de déterminer les spécifications fonctionnelles du filtre à synthétiser, voir figure (7.3) ;

  Constraints and options : ouvre une fenêtre permettant de déterminer les contraints de surface et de test en-ligne ainsi que la technologie cible et l'objet du fichier VHDL qui sera généré (simulation de faute ou prototype), voir aussi figure (7.3) ;

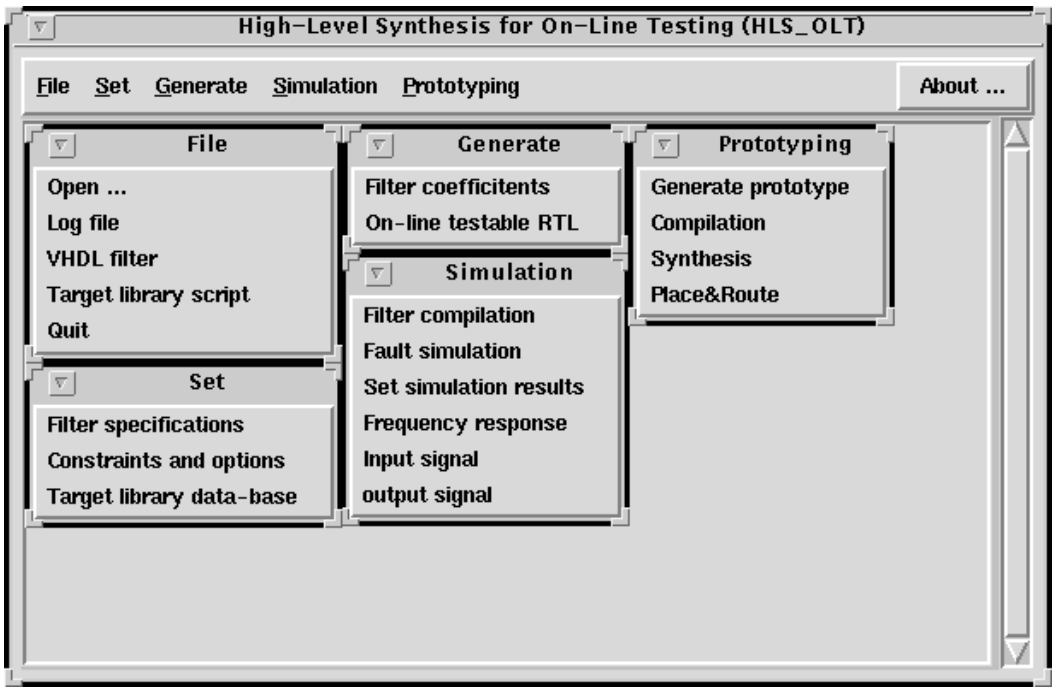


Figure 7.2: *Les différents menus de l'outil.*

Target library data-base : fait tourner un script permettant la génération d'une base de données qui contient la surface, le délai et les vecteurs de test des différentes unités fonctionnelles. Les informations dans cette base de données sont nécessaires pour déterminer l'architecture RTL testable en-ligne du filter selon les contraintes imposées.

- **Generate :**

  Filter coefficients : génére un fichier .m qui est utilisé pour la génération des coefficients du filtre et pour la génération des échantillons d'un signal au choix pour la simulation de faute. La fréquence de ce signal doit être spécifiée avec les spécifications fonctionnelles du filtre ;

  On-line testable RTL : comprend le cœur de l'outil qui permet de réaliser toutes les étapes de la synthèse de haut niveau pour le test en-ligne. Le choix final de l'architecture RTL du filtre ainsi que de la méthode de test en-ligne qui doit être utilisée sont éffectués à cette étape. Le résultat est un fichier VHDL contenant la description RTL testable en-ligne du filtre spécifié.

- **Simulation :**

  Filter compilation : compilation du fichier VHDL obtenu pour permettre de faire la simulation de faute ;

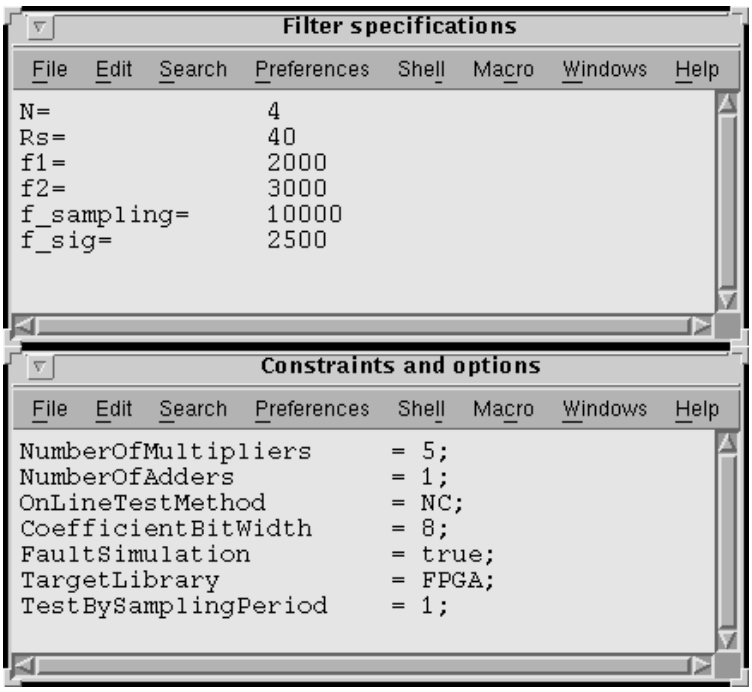


Figure 7.3: *Les spécifications du filtre et les contraintes sur son architecture RTL.*

Fault simulation : lance l'outil vhdldbx de Synopsys pour faire la simulation de faute ;

Set simulation result : réorganise les résultats de la simulation de faute pour les visualiser ;

Frequency response : montre la réponse fréquentielle du filtre telle qu'elle a été générée par matlab ;

Input signal : montre le signal qui est appliqué au filtre lors de la simulation de faute ;

output signal : montre la réponse du filtre au signal d'entrée. Un signal de faute est aussi ajouté pour la détection de faute.

- **Prototyping :**

Generate prototype : génére une description VHDL testable en-ligne pour la programmation d'un FPGA selon les critères imposées par la carte de réalisation ;

Compilation : compilation du prototype généré pour vérifier l'absence d'erreurs ;

Synthesis : réalise la synthèse de bas niveau du prototype avec l'outil leonardo de Mentor Graphics ;

Place&Route : réalise le placement et routage du filtre sur le circuit FPGA. La sortie de cette étape est un fichier binaire à programmer directement sur l'FPGA.

Dans la fenêtre principale de l'outil des message communiqués par les différentes tâches sont affichés pour permettre au utilisateur de suivre l'avancement de l'outil dans la synthèse du filtre. Figure (7.4) montre un exemple de ce message après avoir invoqué la Simulation : Filtre compilation.

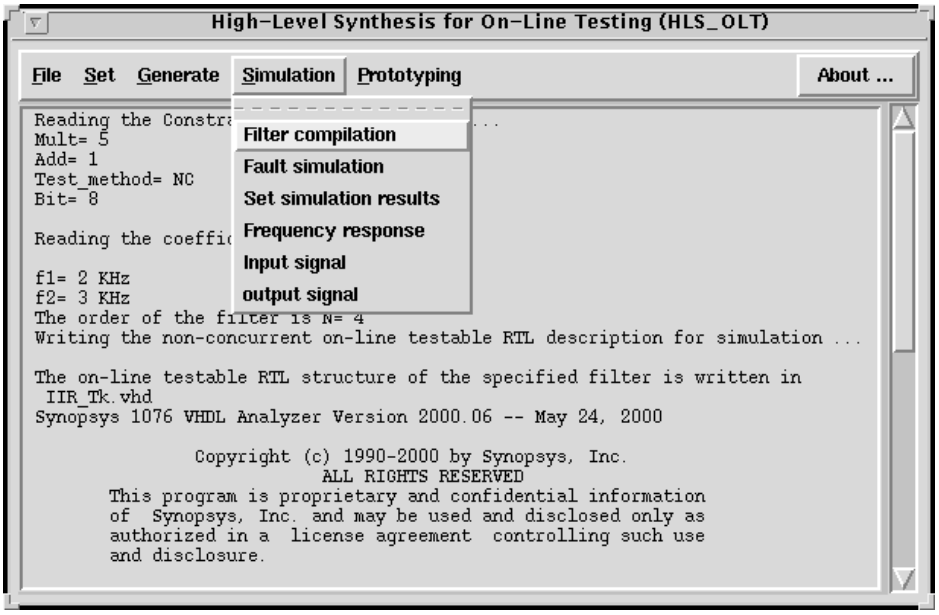


Figure 7.4: *L'interface graphique de l'outil.*

Chapitre 8

Application

Le flot de synthèse de haut niveau pour le test en-ligne qui a été développé tout au long de cette étude sera illustré dans ce chapitre par un exemple de filtre numérique.

Un filtre IIR elliptique de quatrième ordre est utilisé comme exemple d'implémentation. Les spécifications fonctionnelles du filtre sont données sous deux formes. La première forme est un fichier qui contient les différents paramètres du filtre, les coefficients des équations d'état et ceux de la fonction de transfert. La deuxième forme est une description VHDL comportementale par ses équations d'état, voir annexe A. Nous limitons l'espace de solutions aux filtres numériques récursifs d'ordre jusqu'à $N = 10$.

La première tâche à réaliser et la compilation des spécifications fonctionnelles en graphe de flot de données ordonnancé. Les différentes structures de réalisation sont explorées et une méthode de test en-ligne est considérée.

La deuxième tâche est la vérification de la conformité du graphe de flot de données ordonnancé qui a été généré et la détermination finale de la méthode de test en-ligne.

Ensuite, la méthode de test en-ligne retenue est intégrée au système au niveau ordonnancement. Finalement l'architecture testable en-ligne au niveau RTL est générée.

Ces différentes étapes seront détaillées dans la suite de ce chapitre. La solution génétique au problème de compilation et ordonnancement sera améliorée par sa combinaison avec une heuristique adaptée à ce type de systèmes numériques.

8.1 Compilation et ordonnancement

La solution au problème de compilation et ordonnancement a été présentée en chapitre 6. Ce type de solutions est robuste et adapté aux problèmes très complexes d'optimisation. Il reste cependant une approche probabiliste qui nécessite le réglage des différents paramètres pour obtenir de bons résultats. Ce réglage n'est pas en soi une tâche facile. Il nécessite l'application de l'algorithme génétique sur plusieurs expérimentaux afin de choisir les bonnes valeurs des paramètres.

Pour contourner ce type de problèmes, il convient de combiner l'approche génétique avec une approche traditionnelle, telle que la programmation linéaire, ou avec une heuristique adaptée au problème. Cela permet de profiter des avantages que présentent les deux approches. Dans notre cas, l'approche génétique sera combinée avec une heuristique.

L'idée consiste à évaluer les individus de la population initiale. Si un ou plusieurs individus satisfont aux contraintes, le meilleur individu est sélectionné. Si aucun individu ne

représente pas une solution satisfaisante, une exploration exhaustive des solutions voisines de la population initiale est commencée. Cette exploration consiste à jouer sur les valeurs qui dépassent les contraintes imposées. Les contraintes sont imposées soit par le concepteur soit des contraintes technologiques comme par exemple la fréquence opérationnelle maximale. Quatre cas de figures sont à considérer dans ce contexte.

Le premier cas suppose que les contraintes imposées par le concepteur sont satisfaites mais la valeur de la fréquence opérationnelle dépasse la capacité de la cible technologique. deux solutions existent pour ce problème, ce sont :

1. une autre cible technologique doit être choisie ;
2. le nombre de pas de contrôle de l'intervalle fonctionnel doit être diminué par l'augmentation du parallélisme dans l'architecture jusqu'à concurrence de ressource disponible.

Le deuxième cas suppose que les contraintes de surface, sous forme de nombre de multiplieurs, ne sont pas respectées. Dans ce cas, l'architecture est sérialisée jusqu'à concurrence de l'une des contraintes de délai, de test en-ligne ou de capacité technologique.

Le troisième cas suppose que les contraintes de délai ne sont pas respectées. Dans ce cas, deux solutions peuvent être appliquées. Elles consistent à l'augmentation :

1. du parallélisme dans l'architecture jusqu'à concurrence de ressources disponibles ;
2. de la fréquence opérationnelle jusqu'à concurrence de la capacité technologique.

Le quatrième cas suppose que les contraintes de test en-ligne ne sont pas respectées. Deux solutions possibles consistent à l'augmentation de l'intervalle oisif par :

1. l'augmentation de la fréquence opérationnelle jusqu'à concurrence de la capacité technologique ;
2. l'augmentation du parallélisme dans l'architecture jusqu'à concurrence de ressources disponibles.

L'algorithme génétique en figure (6.23) est modifié pour intégrer cette heuristique. Sa nouvelle forme est présentée en figure (8.1). Le problème avec cette heuristique est qu'une solution optimale ne peut pas être garantie et que la complexité de cette exploration augmente avec l'augmentation de l'espace de solutions. Cependant, cette heuristique convient au type de système choisi pour l'application en raison de la limitation aux filtres numériques récursifs (IIR) et de la limitation de l'ordre de filtre IIR à n= 10.

Pour notre exemple d'application, la population initiale sera générée en considérant les contraintes et les éléments suivants:

- le cible technologique est la technologie 0.6 μm, cub de AMS ;
- le nombre de multiplieurs au niveau RTL est limité à $N_{Mult} = 5$ multiplieurs ;
- la latence de faute à satisfaire est $L_{faute} = 0.99$ de la période d'échantillonnage ;
- la méthode de test en-ligne non-concurrent est imposée, $M_{test} = NC$.

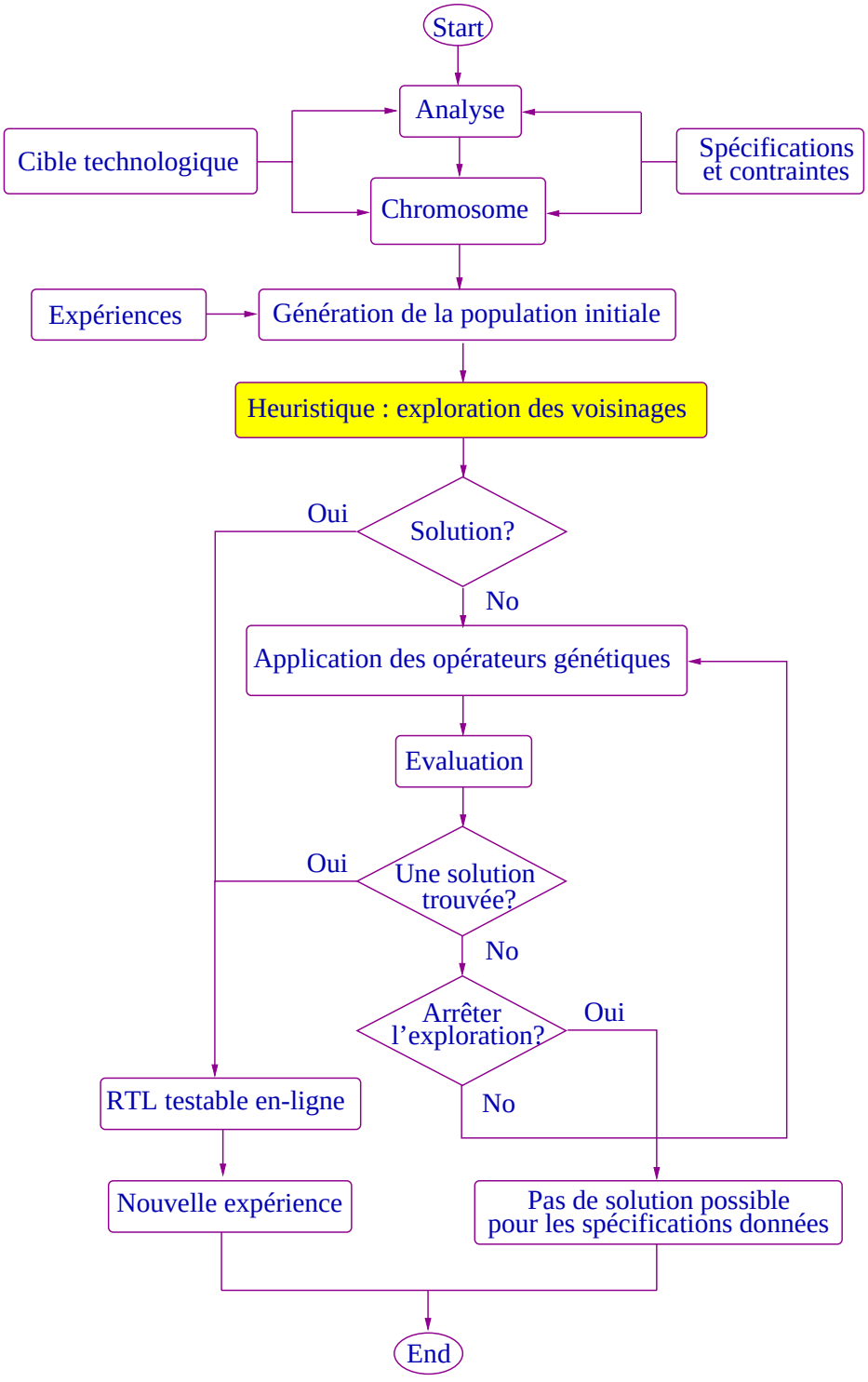


Figure 8.1: *L'heuristique intégrée à l'algorithme génétique.*

8.1.1 Etablissement du chromosome

Les éléments de la partie obligatoire du chromosome sont établis à partir des spécifications fonctionnelles. L'ordre du filtre N= 4, la fréquence d'échantillonnage F_e= 0.5 MHz et la largeur en bits des multiplieurs B= 16 bits.

Le nombre de pas de contrôle V_M qui doivent être disponibles pour le test en-ligne afin de satisfaire aux contraintes de test en-ligne est calculé à partir de la latence de faute demandée et le nombre de vecteurs de test à appliquer sur les multiplieurs V_{Mult}. Ce nombre de vecteurs de test est disponible dans la base de données qui contient les caractéristiques des unités fonctionnelles. Pour la cible technologique choisie, 0.6 μ cub de AMS, on trouve $V_{Mult} = 59$ vecteurs. Nous avons une latence de faute imposée au design final $L_{faute} = 0.99$ de périodes d'échantillonnage. Cela donne :

$$V_M = \frac{V_{Mult}}{L_{faute}} = \frac{59}{0.99} = 59.60 = 60 \; vecteurs$$

A noter que le nombre de temps oisifs disponibles dans l'intervalle fonctionnel n'a pas été considéré en raison de l'indisponibilité, jusqu'à cette étape, d'informations autour l'architecture au niveau graphe de flot de données ordonnancé.

8.1.2 Population initiale

La population initiale est représentée en figure (8.2). Les graphes de flot de données ordonnancés qui correspondent à ces individus sont présentés en annexe B.

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}
Forme directe 1	Architecture série	4	0.5	16	60	1	D1	NC	10	38.0
	Architecture parallèle	4	0.5	16	60	7	D1	NC	5	32.5
	Nouvelle architecture	4	0.5	16	60	5	D1	NC	5	32.5
Forme directe 2	Architecture série	4	0.5	16	60	1	D2	NC	10	35.0
	Architecture parallèle	4	0.5	16	60	7	D2	NC	4	32.0
	Nouvelle architecture	4	0.5	16	60	5	D2	NC	4	32.0
Equation d'état	Architecture série	4	0.5	16	60	1	EE	NC	26	43.0
	Architecture parallèle	4	0.5	16	60	16	EE	NC	4	32.0
	Architecture intermédiaire	4	0.5	16	60	5	EE	NC	8	34.0
	Nouvelle architecture	4	0.5	16	60	5	EE	NC	8	34.0

Figure 8.2: *population initiale.*

8.1.3 Fonction d'évaluation

Nous allons, maintenant, construire le chromosome référence qui va nous servir pour le calcul de la *fitness* de chaque individu.

La première partie contient l'ordre du filtre N, la fréquence d'échantillonnage F_e et la largeur en bits des multiplieurs B. Cette partie est établie directement des spécifications fonctionnelles du filtre.

La deuxième partie contient les limites imposées au design final comme des contraintes. Il s'agit du nombre de pas de contrôle qui doivent être disponibles pour le test en-ligne V_{M_ref}, du nombre de multiplieurs disponibles au niveau RTL N_{Mult_ref} et de la fréquence opérationnelle maximale F_{Op_ref} que permet le cible technologique. Cette limite technologique avec la valeur de la fréquence d'échantillonnage F_e et la valeur de V_M déterminent le nombre maximal de pas de contrôle de l'intervalle fonctionnel.

Dans notre cas, le délai du multiplieur de la cible technologique est de 28.68 ns. Cela donne la valeur maximale de la fréquence opérationnelle $F_{Op} = 34.9$ MHz. Le nombre maximal de pas de contrôle de l'intervalle fonctionnel est dans ce cas :

$$N_{ctrl\_ref} = \frac{F_{Op\_ref}}{F_e} - V_{M\_ref} = \frac{34.9}{0.5} - 60 = 9.8 = 9 \text{ pas de contrôle}$$

Le chromosome référence a donc la forme suivante :

N	F_e	B	V_{M_ref}	N_{Mult_ref}	S	M_{test}	N_{ctrl_ref}	F_{Op_ref}
4	0.5	16	60	5	S	NC	30	34.9

8.1.4 Evaluation de la population initiale

Les individus de la population initiale seront évalués pour en sélectionner ceux qui représentent une solution satisfaisante aux contraintes imposées. Cette évaluation se fait par le calcul des trois fonctions de *fitness* : f_{VM}, f_{Mult} et f_{ctrl} (voir la section 6.7.3).

Les contraintes de test en-ligne sont toujours satisfaites en raison de la valeur de V_M qui est déterminée à partir de ces contraintes. L'évaluation de la *fitness* des individus revient donc au calcul des fonctions d'évaluation des contraintes de surface et de délai. En figure (8.3), la population initiale est représentée avec les valeurs de la *fitness* pour chaque individu.

L'individu qui représente la nouvelle architecture des équations d'état est éliminé de la population car il est répété. Le résultat de cette évaluation est que trois individus répondent aux contraintes imposées et ils peuvent être sélectionnés comme une solution pour l'architecture du filtre.

Une comparaison entre ces trois solutions montre que l'individu qui représente la nouvelle architecture de la forme directe 2 est la meilleure solution car il donne le minimum au niveau de nombre de pas de contrôle de l'intervalle fonctionnel et donc le minimum de fréquence opérationnelle.

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}	f_{VM}	f_{Mult}	f_{ctrl}
Forme directe 1	Architecture série	4	0.5	16	60	1	D1	NC	10	38.0	1	4	0.5
	Architecture parallèle	4	0.5	16	60	7	D1	NC	5	32.5	1	0.33	4
	Nouvelle architecture	4	0.5	16	60	5	D1	NC	5	32.5	1	1	4
Forme directe 2	Architecture série	4	0.5	16	60	1	D2	NC	10	35.0	1	4	0.5
	Architecture parallèle	4	0.5	16	60	7	D2	NC	4	32.0	1	0.33	5
	Nouvelle architecture	4	0.5	16	60	5	D2	NC	4	32.0	1	1	5
Equation d'état	Architecture série	4	0.5	16	60	1	EE	NC	26	43.0	1	4	0.06
	Architecture parallèle	4	0.5	16	60	16	EE	NC	4	32.0	1	0.09	5
	Architecture intermédiaire	4	0.5	16	60	5	EE	NC	8	34.0	1	1	1
	Nouvelle architecture	4	0.5	16	60	5	EE	NC	8	34.0	1	1	1

Figure 8.3: *Evaluation de la population initiale.*

8.1.5 Optimisation

Deux approches sont possibles pour l'optimisation de la solution obtenue. Le principe se repose sur l'exploration de l'espace de solutions. Il est possible de partir de n'importe quelle solution pour trouver la solution optimale. Cependant, le coût en terme de temps d'exploration et de ressource à impliquer ne sera pas le même. La première approche consiste à paralléliser une architecture série par l'ajout d'un multiplieur au niveau RTL. La deuxième approche consiste à sérialiser une architecture parallèle par l'enlèvement d'un multiplieur au niveau RTL. Il est encore possible d'utiliser les deux approches pour la recherche d'une solution optimale au voisinage des individus de la population initiale.

D'autres solutions peuvent être trouvées au voisinage de plusieurs individus de la population initiale. Ces solutions peuvent être meilleures des solutions de la population initiale. Nous allons considérer, à titre d'exemple, les individus qui représentent les architectures séries et les nouvelles architectures de toutes les structures de réalisation considérées. Un parallélisme sera introduit aux architectures série par l'ajout d'un deuxième multiplieur. De l'autre côté, une sérialisation sera introduite aux nouvelles architectures par l'enlèvement d'un multiplieur. D'autres solutions sont ainsi obtenues. L'économie en surface atteint 50% pour les architectures à deux multiplieurs. Les nouvelles solutions sont présentées en figure (8.4). Les architectures qui correspondent à ces solutions sont présentées en annexe B sous forme de graphes de flot de données ordonnancés.

Cette exploration a pu être fait facilement en raison de la simplicité relative du problème considéré. L'espace de solutions augmente considérablement quand l'ordre du filtre atteint quelques centaines ou quand il s'agit d'une architecture plus complexe composée de plusieurs

		N	F_e	B	V_M	N_{Mult}	S	M_{test}	N_{ctrl}	F_{Op}	f_{VM}	f_{Mult}	f_{ctrl}
Forme directe 1	Deux multiplieurs	4	0.5	16	60	2	D1	NC	7	33.5	1	3	0.33
	Quatre multiplieurs	4	0.5	16	60	4	D1	NC	5	32.5	1	1	4
Forme directe 2	Deux multiplieurs	4	0.5	16	60	2	D2	NC	7	33.5	1	2	0.33
	Quatre multiplieurs	4	0.5	16	60	4	D2	NC	4	32.0	1	1	5
Equation d'état	Deux multiplieurs	4	0.5	16	60	2	EE	NC	17	43.5	1	4	0.11
	Quatre multiplieurs	4	0.5	16	60	4	EE	NC	8	34.0	1	1	1

Figure 8.4: *Optimisation par exploration des solutions voisines de celles de la population initiale.*

filtres. D'où l'utilité de l'utilisation des algorithmes génétiques pour résoudre ce type de problèmes.

8.2 Implémentation du test en-ligne non-concurrent

Cette section a comme objectif de présenter une implémentation d'une des solutions qui ont été obtenues pour le filtre numérique étudié. Il s'agit de réaliser en description VHDL structurelle une des architectures parallèles du filtre qui est l'architecture intermédiaire des équations d'état. Le circuit de test en-ligne est intégré au filtre et la simulation de faute est effectuée. Une unité de MAC testable en-ligne par la méthode de test en-ligne non-concurrent est aussi présentée à titre illustrative. D'autres simulation de fautes peuvent être trouvées en annexe D.

8.2.1 Choix d'intégration du circuit de test en-ligne

Deux façons sont possibles pour l'intégration du circuit de test en-ligne au système. La première consiste à utiliser un seul circuit de test en-ligne pour tous les opérateurs du même type. La deuxième consiste à intégrer le circuit de test en-ligne à chaque unité fonctionnelle de manière à disposer directement d'unités fonctionnelles testables en-ligne dans une bibliothèque de modules. Nous allons évaluer ces deux méthodes d'intégration du circuit de test en-ligne pour l'exemple du filtre étudié dans ce chapitre.

Considérons la réalisation du filtre par cinq architectures ; l'architecture série et quatre architectures parallèles. Les architectures parallèles sont celles qui contiennent 2, 3, 4 et 5 multiplieurs respectivement dans le chemin de données au niveau RTL. L'évaluation du coût en surface de la partie analyse de signature et détection de faute du circuit de test en-ligne est faite en figure (8.5).

On peut constater que le coût de n circuits réalisés pour un seul multiplieur est inférieur au coût d'un circuit réalisé pour n multiplieurs. C'est à dire que l'intégration du circuit de test en-ligne à chaque unité fonctionnelle revient plus économique en surface que la conception d'un seul circuit pour plusieurs multiplieurs. En effet, ce choix est encore plus efficace pour les raisons suivantes :

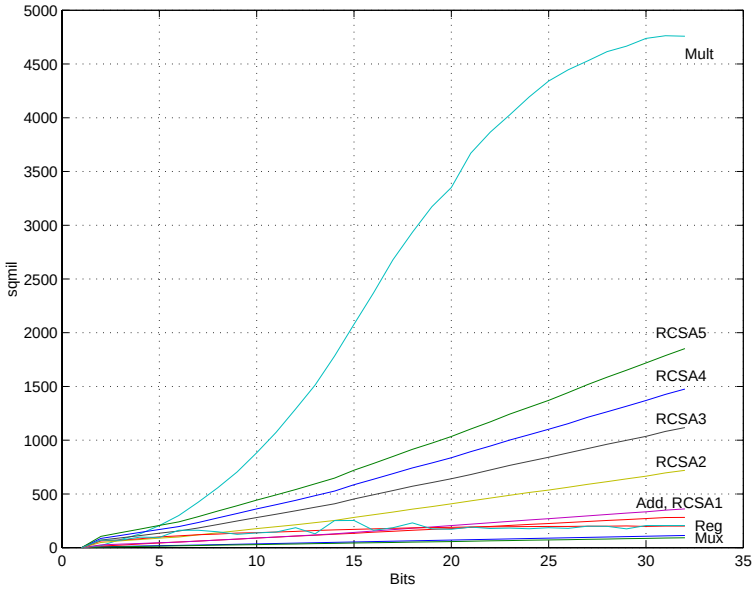


Figure 8.5: *Surcoût en surface de plusieurs configurations du circuit de test en-ligne non-concurrent, technologie 0.6 μ cub de AMS.*

- Economie en temps de conception par le fait de disposer d'unités fonctionnelles testables en-ligne avec leurs circuits intégrés de test en-ligne ;
- Economie additionnelle en surface et en consommation par le fait de supprimer le besoin de routage des vecteurs de test et des réponses des unités fonctionnelles pour un circuit central de test en-ligne ;
- Amélioration des performances du circuit de test en-ligne par l'accélération de l'accès aux unités fonctionnelles.

Cela nous mène au choix de l'intégration circuit de test en-ligne à chaque unité fonctionnelle.

8.2.2 Simulation de faute de l'architecture parallèle

La figure (8.6) représente le chemin de données avec la partie contrôle de l'architecture intermédiaire. Le circuit de test en-ligne est intégré à chaque multiplieur mais représenté sur le schéma par deux boîtes seulement pour simplifier. La première boîte contient la partie génération de vecteurs de test pour tous les multiplieurs alors que la deuxième boîte contient la partie de vérification de réponse et génération de signal d'erreur.

Cette architecture a été simulée avec l'outil *vhdldbx* de SYNOPSYS. L'injection de faute a été simulée par la génération de collage à 1 ou à 0 sur un des bits des entrées/sorties des multiplieurs. Deux types d'erreurs sont choisis pour illustrer la simulation de faute. Le premier type illustre l'erreur qui n'apporte pas une grande perturbation à la réponse du filtre, figure (8.7). Cette erreur est générée par le collage à 0 du bit 12 de l'entrée B du

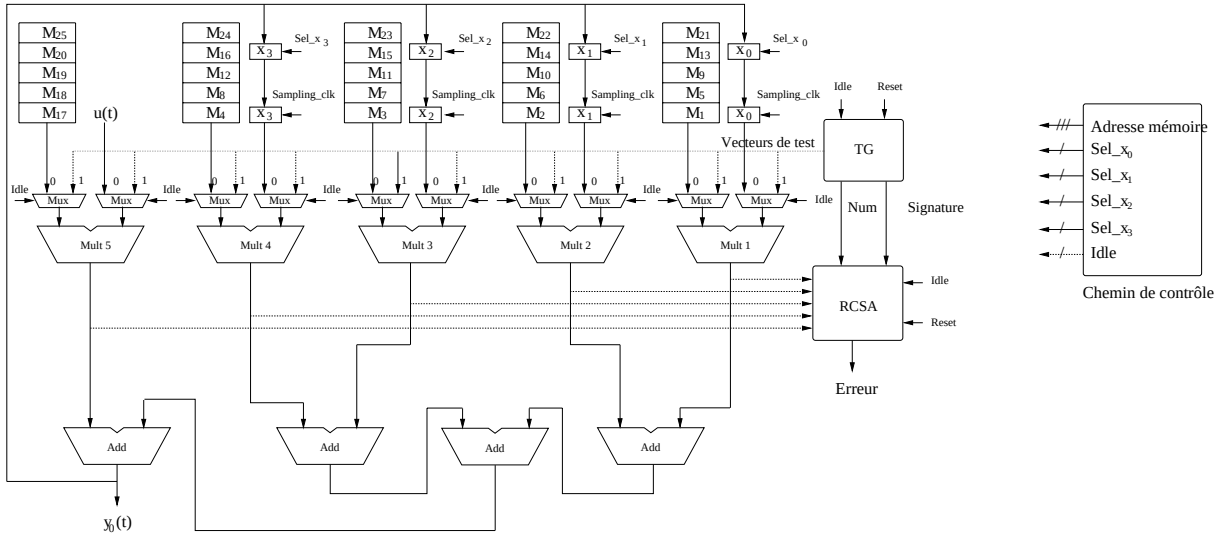


Figure 8.6: *Le chemin de données de l'architecture intermédiaire testable en-ligne avec la méthode de test en-ligne non-concurrent.*

multiplieur $Mult_1$. Le deuxième type illustre l'erreur qui perturbe complètement la réponse du filtre, figure (8.8). Cette erreur est générée par le collage à 1 du bit 14 de l'entrée B du multiplieur $Mult_1$.

Dans ces deux types d'erreurs, la faute a été injectée à l'instant 1.2×10^5 ns. Le signal de détection d'erreur est activé à chaque instant qu'un vecteur de test excite la faute. Pour les vecteurs de test qui n'excitent pas la faute le signale d'erreur est remise à zéro. Cela permet d'évaluer statistiquement la couverture des fautes par les différents vecteurs de test.

Le mécanisme de la détection de faute par le test en-ligne non-concurrent est présenté en annexe C.

8.2.3 Simulation de faute de l'architecture série

Il s'agit de présenter une autre architecture de réalisation utilisant un seul multiplieur et un seul additionneur (unité de MAC).

La figure (8.9) représente le chemin de données avec la partie contrôle de l'unité de MAC. Le circuit de test en-ligne est intégré au multiplieur.

Cette architecture a été simulée avec l'outil *vhdldbx* de SYNOPSYS. L'injection de faute a été simulée par la génération de collage à 1 ou à 0 sur un des bits des entrées/sorties du multiplieur. Deux types d'erreurs sont choisis pour illustrer la simulation de faute. Le premier type illustre l'erreur qui n'apporte pas une grande perturbation à la réponse du filtre, figure (8.10). Cette erreur est générée par le collage à 0 du bit 2 de l'entrée B du multiplieur. Le deuxième type illustre l'erreur qui perturbe complètement la réponse du filtre, figure (8.11). Cette erreur est générée par le collage à 0 du bit 12 de l'entrée B du multiplieur $Mult_1$.

Dans ces deux types d'erreurs, la faute a été injectée à l'instant 1×10^5 ns. Le signal de

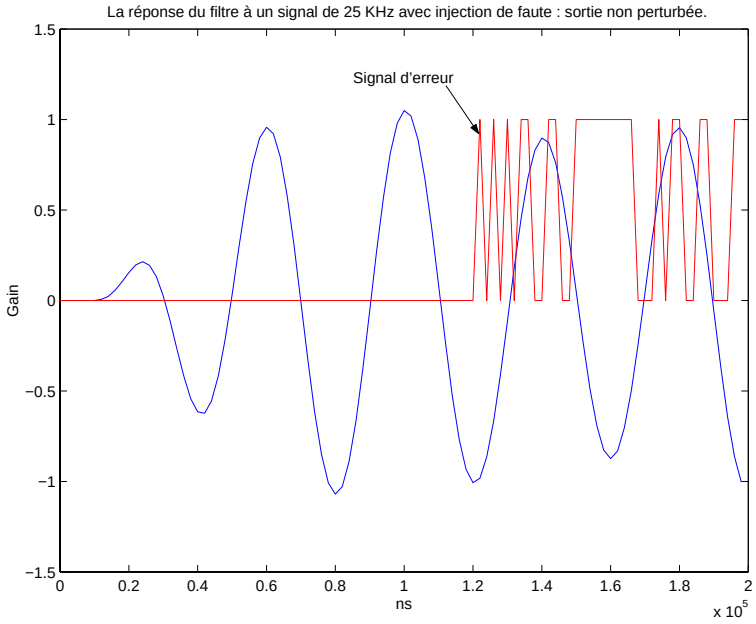


Figure 8.7: *Simulation de faute du test en-ligne non-concurrent : collage à 0 du bit 12 de l'entrée B du* $Mult_1$.

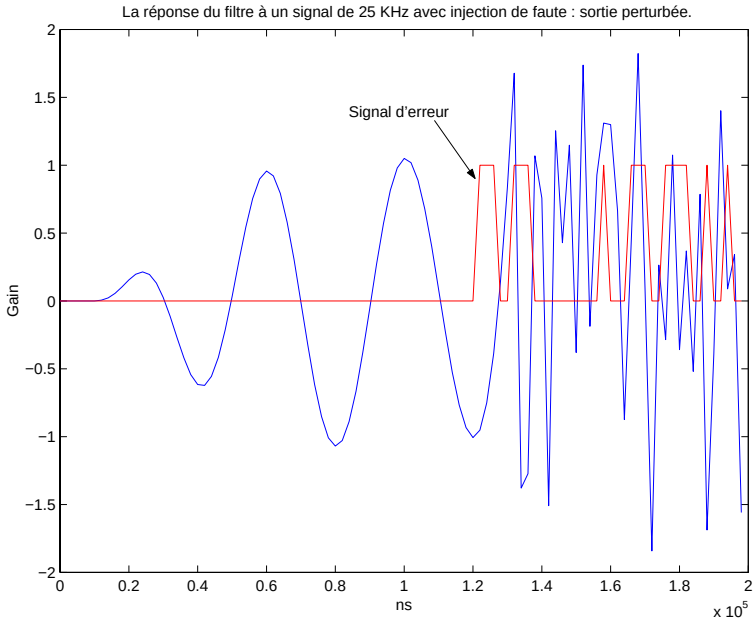


Figure 8.8: *Simulation de faute du test en-ligne non-concurrent : collage à 1 du bit 14 de l'entrée B du* $Mult_1$.

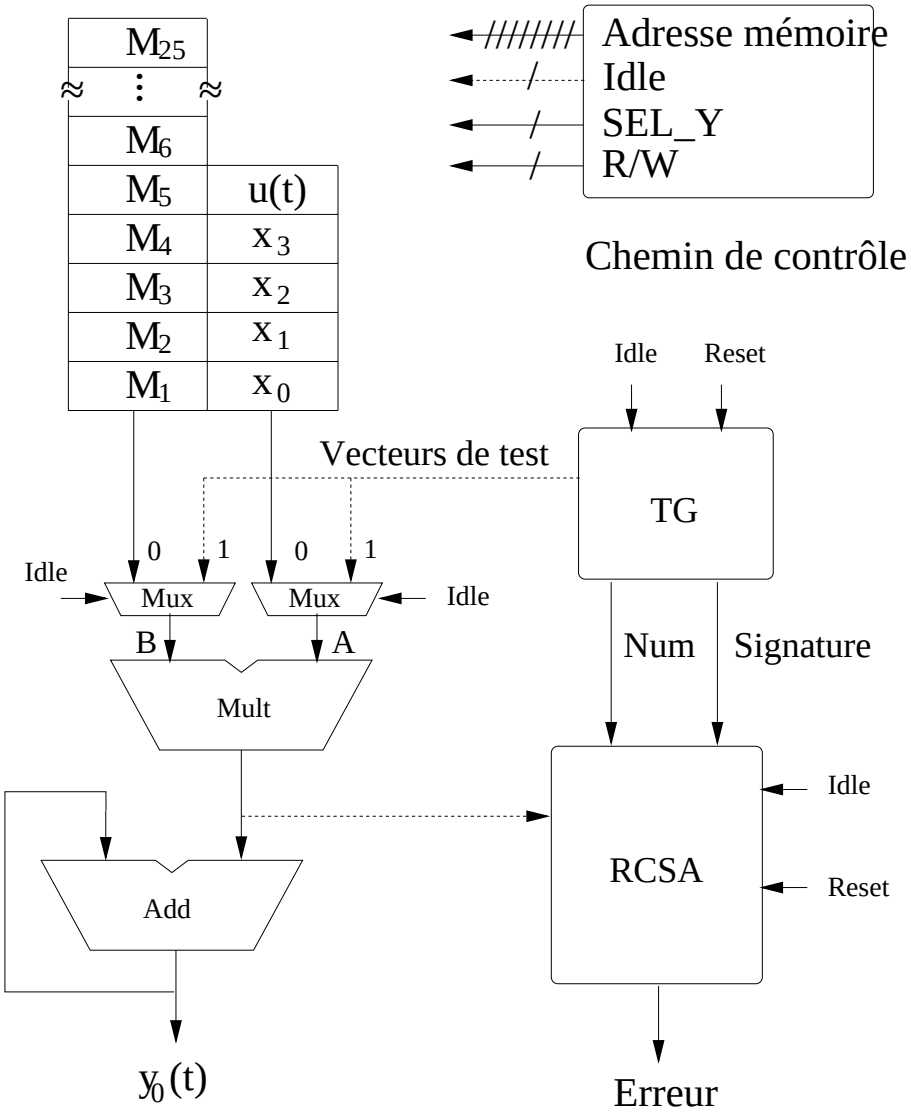


Figure 8.9: *Le chemin de données de l'unité de MAC testable en-ligne par la méthode de test en-ligne non-concurrent.*

détection d'erreur est activé à chaque instant qu'un vecteur de test excite la faute. Pour les vecteurs de test qui n'excitent pas la faute le signale d'erreur est remise à zéro. Cela permet d'évaluer statistiquement la couverture des fautes par les différents vecteurs de test.

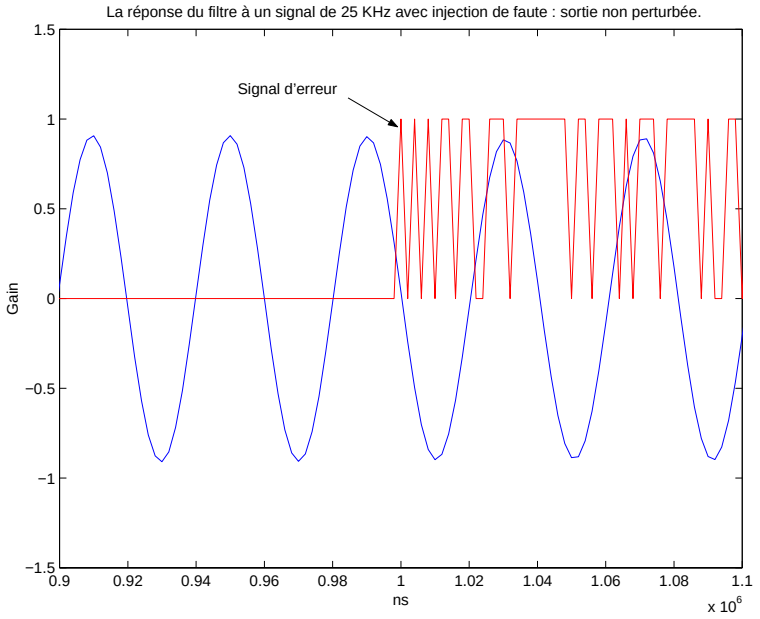


Figure 8.10: *Simulation de faute du test en-ligne non-concurrent : collage à 0 du bit 2 de l'entrée B du multiplieur.*

8.3 Implémentation du test en-ligne semi-concurrent

Le même filtre qui a été étudié dans les sections précédentes sera implémenté dans cette section avec la méthode de test en-ligne semi-concurrent. La simulation de fautes est effectuée sur les architectures présentées. D'autres simulation de fautes peuvent être trouvées en annexe D.

8.3.1 Simulation de faute de l'architecture parallèle

La figure (8.12) représente le chemin de données avec la partie contrôle de l'architecture intermédiaire. L'insertion du graphe de test en-ligne est effectuée par la modification de la partie contrôle pour refaire le calcul nominal dans les temps oisifs des unités fonctionnelles. La permutation des unités fonctionnelles est réalisée par la reconfiguration du chemin de données avec des multiplexeurs.

Cette architecture a été simulée avec injection de faute. La première simulation considère le collage á 0 du bit 2 de l'entrée B de $Mult_1$, figure (8.13). La réponse du filtre n'est perturbée et la faute est détectée. La deuxième simulation considère le collage à 1 du bit 14 de la même entrée du multiplieur, figure (8.14). La réponse du filtre est complètement perturbée et la faute est détectée. La troisième simulation consiste à injecter un collage à 0 au bit 12, figure (8.15). La réponse du filtre est perturbée alors que la faute n'est pas détectée.

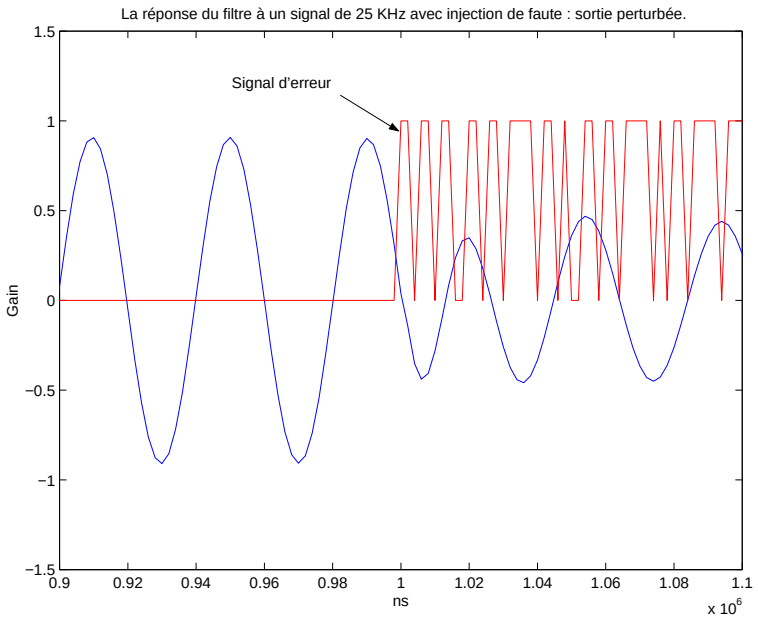


Figure 8.11: *Simulation de faute du test en-ligne non-concurrent : collage à 0 du bit 12 de l'entrée B du multiplieur.*

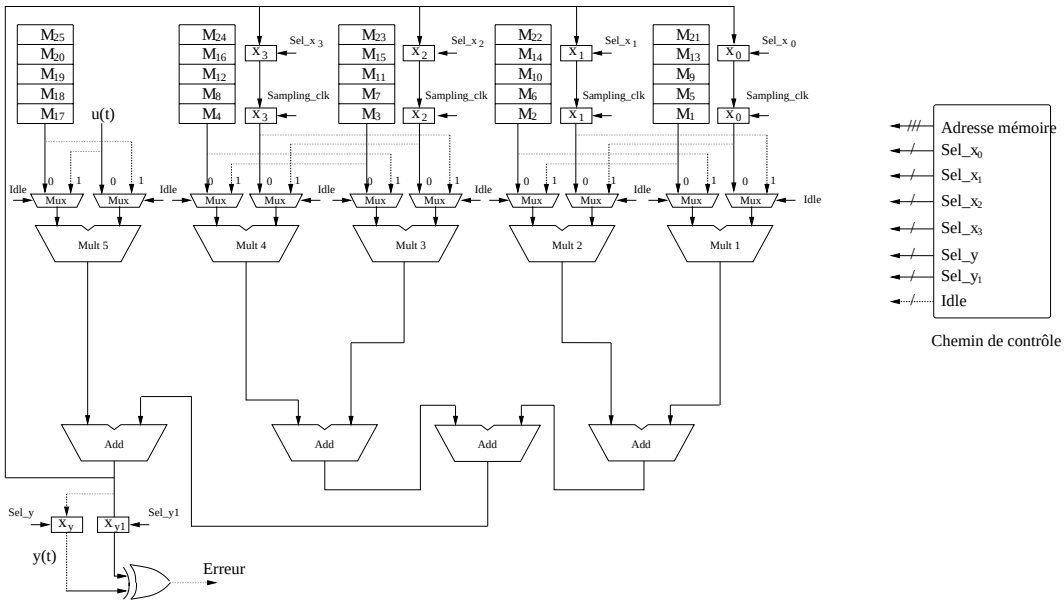


Figure 8.12: *Le chemin de données de l'architecture intermédiaire testable en-ligne par la méthode de test en-ligne semi-concurrent : insertion du graphe de test en-ligne avec permutation d'unités fonctionnelles.*

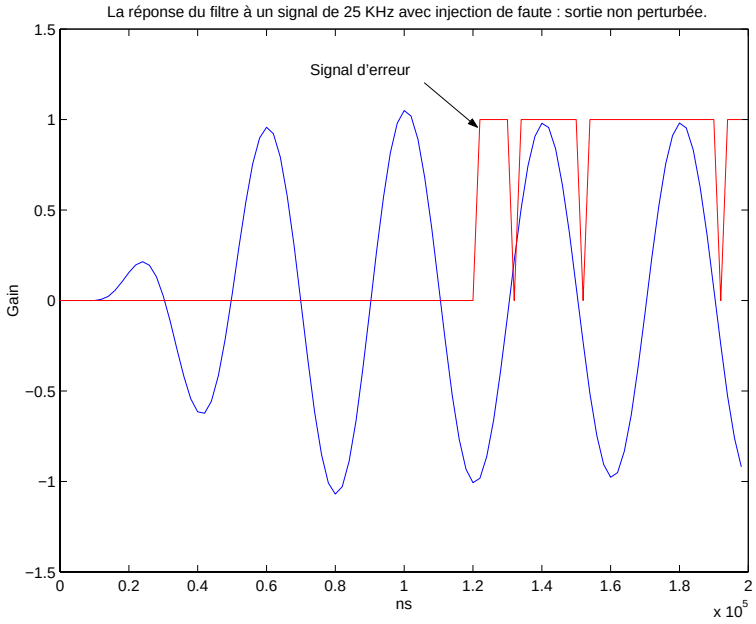


Figure 8.13: *Simulation de faute du test en-ligne semi-concurrent : collage à 0 du bit 2 de l'entrée B de* $Mult_1$.

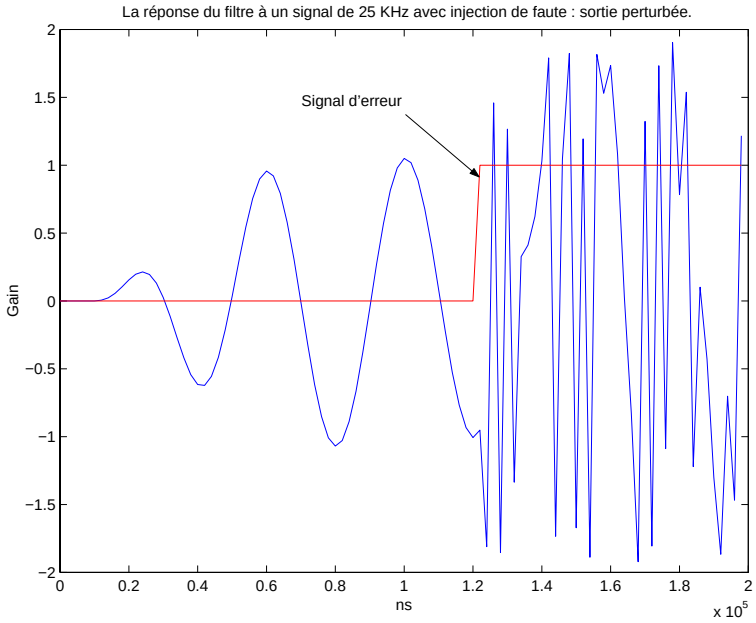


Figure 8.14: *Simulation de faute du test en-ligne semi-concurrent : collage à 1 du bit 14 de l'entrée B de* $Mult_1$.

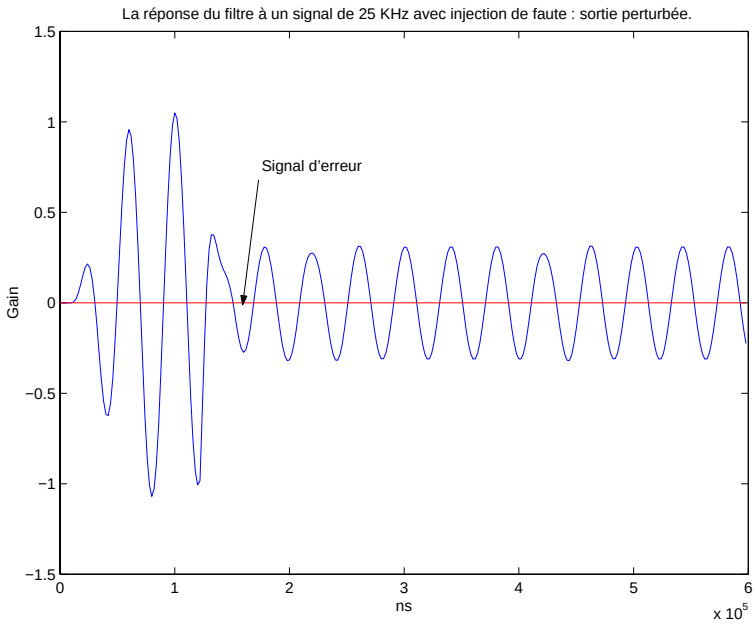


Figure 8.15: *Simulation de faute du test en-ligne semi-concurrent : collage à 0 du bit 12 de l'entrée B de* $Mult_1$.

8.3.2 Simulation de faute de l'architecture série

Il s'agit ici de réaliser une unité de MAC avec la méthode de test en-ligne semi-concurrent. L'insertion d'un graphe de test en-ligne est réalisée au niveau de la partie contrôle et la permutation se fait entre les deux entrées du multiplieur. Le chemin de données testable en-ligne est présenté en figure (8.16). La simulation de faute est présentée en figures(8.17) et (8.18).

8.4 Evaluation et comparaison

Dans cette section, nous allons évaluer le coût additionnel en surface des méthodes de test en-ligne non-concurrent et semi-concurrent pour deux cibles technologiques. La première cible technologique est la technologie 0.6 μ cub de AMS et la deuxième est la technologie 0.35 μ csx de AMS.

Cette évaluation sera réalisée pour plusieurs architectures et pour plusieurs largeurs en bits des multiplieurs. A noter que le circuit de test en-ligne non-concurrent n'implique aucune dégradation en délai. Cela est dû d'une part à l'exploitation des temps oisifs des opérateurs et d'une autre part au fait que le délai de ce circuit de test en-ligne est nettement inférieur au délai du multiplieur ce qui n'affecte pas l'horloge fonctionnelle. De même, le test en-ligne semi-concurrent qui répète le calcul nominal dans les temps oisifs des opérateurs n'implique aucune dégradation en délai en raison de l'utilisation des même opérateurs sur les même données pour le test en-ligne. Seul le test en-ligne semi-concurrent qui exploite la distributivité de l'addition sur la multiplication implique une dégradation à la fois en surface

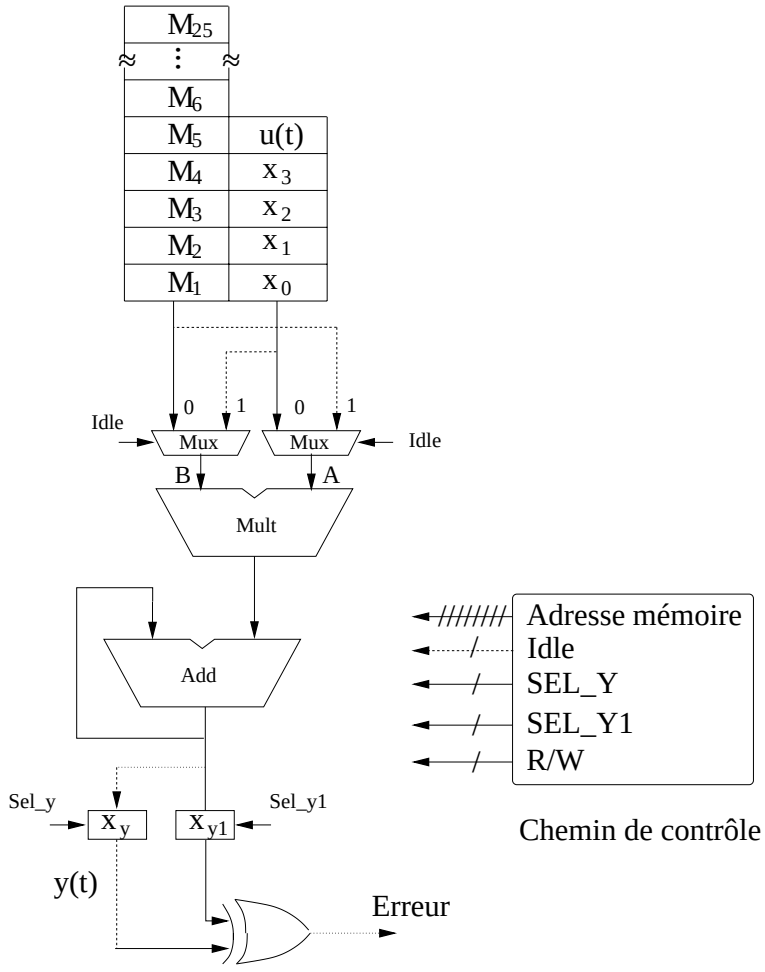


Figure 8.16: *Le chemin de données de l'architecture série testable en-ligne par la méthode de test en-ligne semi-concurrent : insertion du graphe de test en-ligne avec permutation des entrées du multiplieur.*

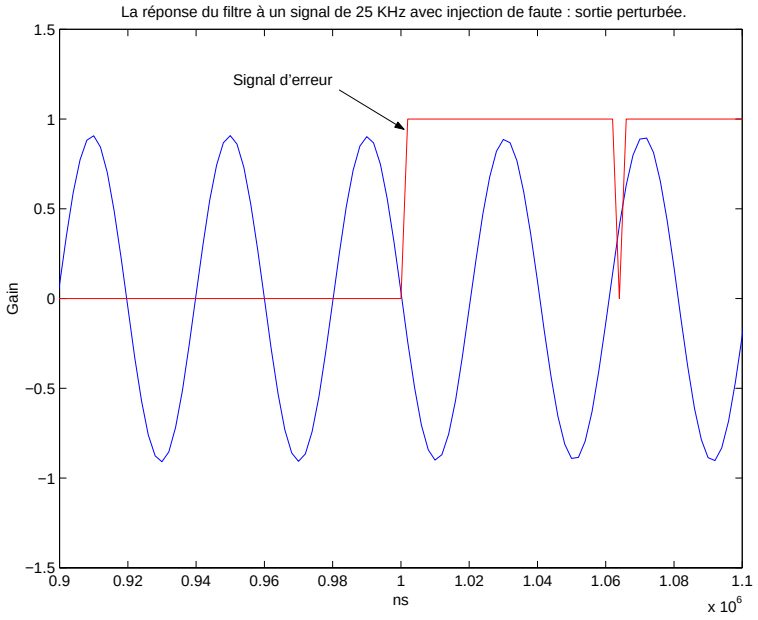


Figure 8.17: *Simulation de faute du test en-ligne semi-concurrent : collage à 0 du bit 2 de l'entrée B de* $Mult_1$.

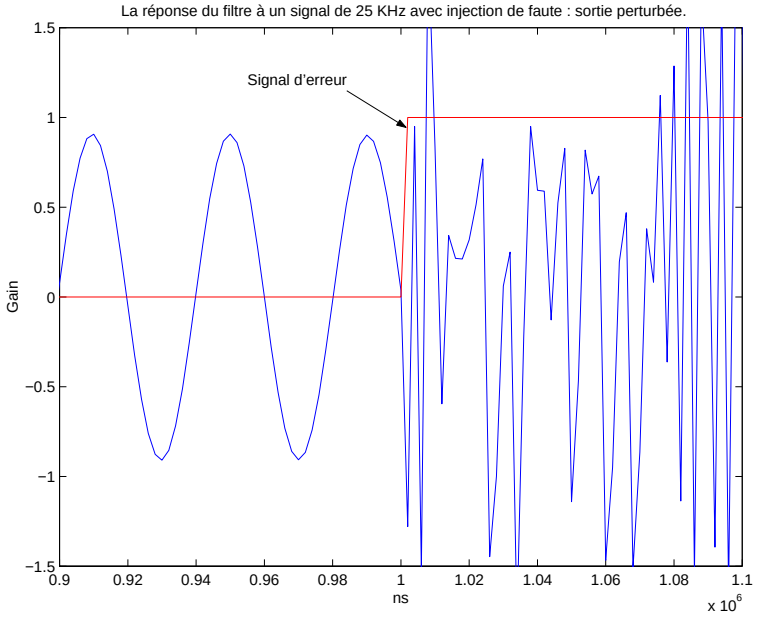


Figure 8.18: *Simulation de faute du test en-ligne semi-concurrent : collage à 1 du bit 14 de l'entrée B de* $Mult_1$.

et en délai. Cette dégradation a été évaluée, au niveau unité fonctionnelle, en chapitre 4, voir la figure (4.6). Nous allons évaluer le surcoût global, au niveau architecture, de la méthode de test non-concurrent et de la méthode de test semi-concurrent avec l'exploitation de la distributivité.

8.4.1 Effet de l'architecture

Dans un premier temps, considérons la réalisation d'un tel filtre par cinq architectures. Ce sont l'architecture série et celles parallèles à 2, 3, 4 et 5 multiplieurs. Le coût additionnel en surface dû au test en-ligne des différentes architectures est présenté en figure (8.19).

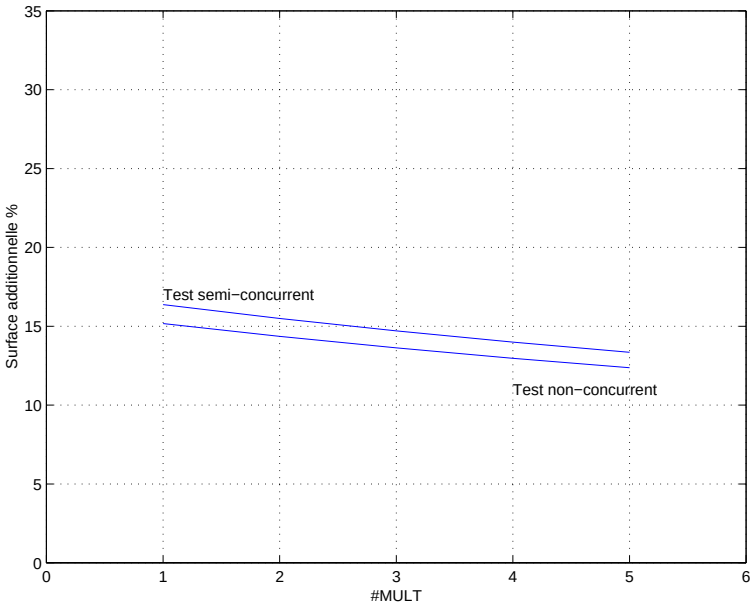


Figure 8.19: *Evaluation du coût additionnel en surface des méthodes de test en-ligne pour une largeur de bit des multiplieurs égale à 16 bits et pour différentes architectures (0.6 μ cub de AMS).*

On peut constater une légère baisse en coût pour les méthodes de test en-ligne non-concurrent et semi-concurrent avec l'augmentation du nombre de multiplieurs. Cela est dû à l'augmentation de la complexité de l'architecture au niveau partie contrôle et communication alors que le circuit de test en-ligne est intégré individuellement à chaque unité fonctionnelle ce qui n'entraîne pas un surcoût en communication. De même, l'augmentation de surcoût en contrôle reste constante avec chaque multiplieur ajouté.

Par ailleurs, le surcoût en surface de la méthode de test en-ligne semi-concurrent est plus important que celui de la méthode de test en-ligne non-concurrent. Cela revient à la dégradation importante en surface des multiplieurs lors du passage du nombre de bit B à B+1. Cette dégradation peut largement dépasser la surface du circuit de test en-ligne non-concurrent.

8.4.2 Effet de la largeur en bit des unités fonctionnelles

Dans un deuxième temps, considérons la réalisation d'un filtre par l'architecture série mais avec différentes largeurs en bit des multiplieurs. Le coût additionnel en surface dû au test en-ligne est présenté en figure (8.20).

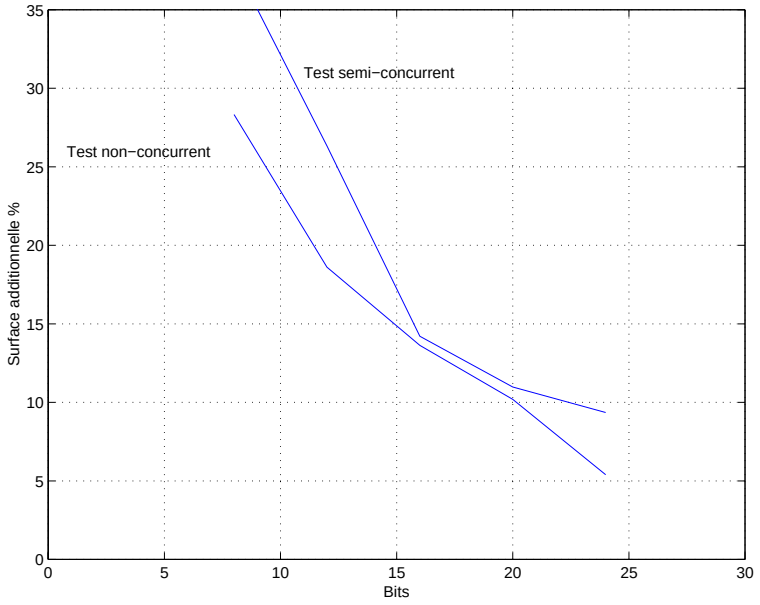


Figure 8.20: *Evaluation du coût additionnel en surface des méthodes de test en-ligne pour l'architecture série et pour différentes largeurs en bit du multiplieur (0.6 μ cub de AMS).*

On peut constater ici une baisse très rapide du coût du circuit de test en-ligne non-concurrent et semi-concurrent. Cela est dû à la complexité croissante de façon très rapide pour les multiplieurs, voir figures(3.6) et (4.6), ce qui diminue très rapidement l'effet du circuit de test en-ligne sur l'architecture.

8.4.3 Effet de la cible technologique

Pour terminer cette évaluation des méthodes de test en-ligne non-concurrent et semi-concurrent, nous considérons l'effet du choix de la technologie sur le surcoût en surface du circuit de test en-ligne. La comparaison sera faite entre la technologie 0.6 μ cub de AMS et celle 0.35 μ csx de AMS.

Premièrement, considérons l'évaluation du coût en surface du circuit de test en-ligne non-concurrent réalisé par la technologie 0.35 μ csx de AMS. La figure (8.21) montre que la surface du circuit de test en-ligne est plus important en comparaison avec celle de la technologie 0.6 μ cub de AMS présentée en figure (8.5). En plus, ce coût augmente considérablement si l'on considère un circuit de test en-ligne central pour plusieurs unités fonctionnelles. Cela confirme le choix de l'intégration du circuit de test en-ligne à chaque unité fonctionnelle.

A noter que le coût du circuit de test en-ligne conçu pour un seul multiplieur par cette technologie est nettement supérieur à celui de la technologie 0.6 μ.

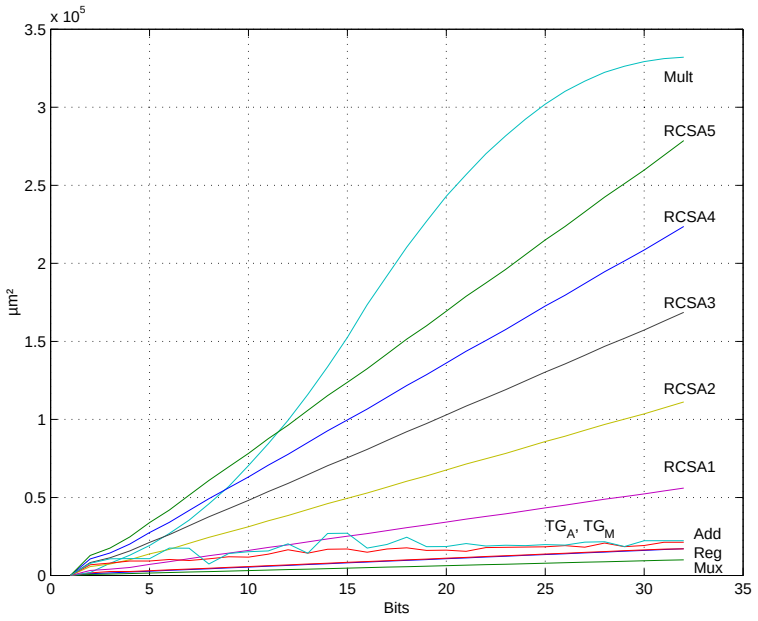


Figure 8.21: *Surcoût en surface de plusieurs configurations du circuit de test en-ligne non-concurrent, technologie 0.35 μ csx de AMS.*

Deuxièmement, nous allons évaluer l'effet du choix de l'architecture et de la largeur en bit des unités fonctionnelles sur le surcoût des méthodes de test en-ligne avec la technologie csx 0.35 μ. L'effet de l'architecture et celui de la largeur en bit des unités fonctionnelles sont présentés en figure (8.22). On peut constater clairement la différence entre les deux technologies en ce qui concerne la méthode de test en-ligne non-concurrent. Le surcoût du circuit de test en-ligne non-concurrent pour la technologie 0.35 μ est nettement supérieur à celui de la technologie 0.6 μ. Cela revient au coût élevé du circuit réalisé par la technologie 0.35 μ, voir la figure (8.21).

Il convient de noter, finalement, que le circuit de test en-ligne pour la méthode de test en-ligne non-concurrent a été synthétisé automatiquement par l'outil design_analyzer de SYNOPSYS à partir d'une description comportementale. Aucune optimisation en full-custom n'a été réalisée pour une ou plusieurs parties du circuit. Ce qui peut expliquer le coût un peu élevé de ce circuit. Une optimisation en full-custom d'une ou plusieurs parties du circuit peut permettre d'économiser entre 20% et 30% de la surface du circuit.

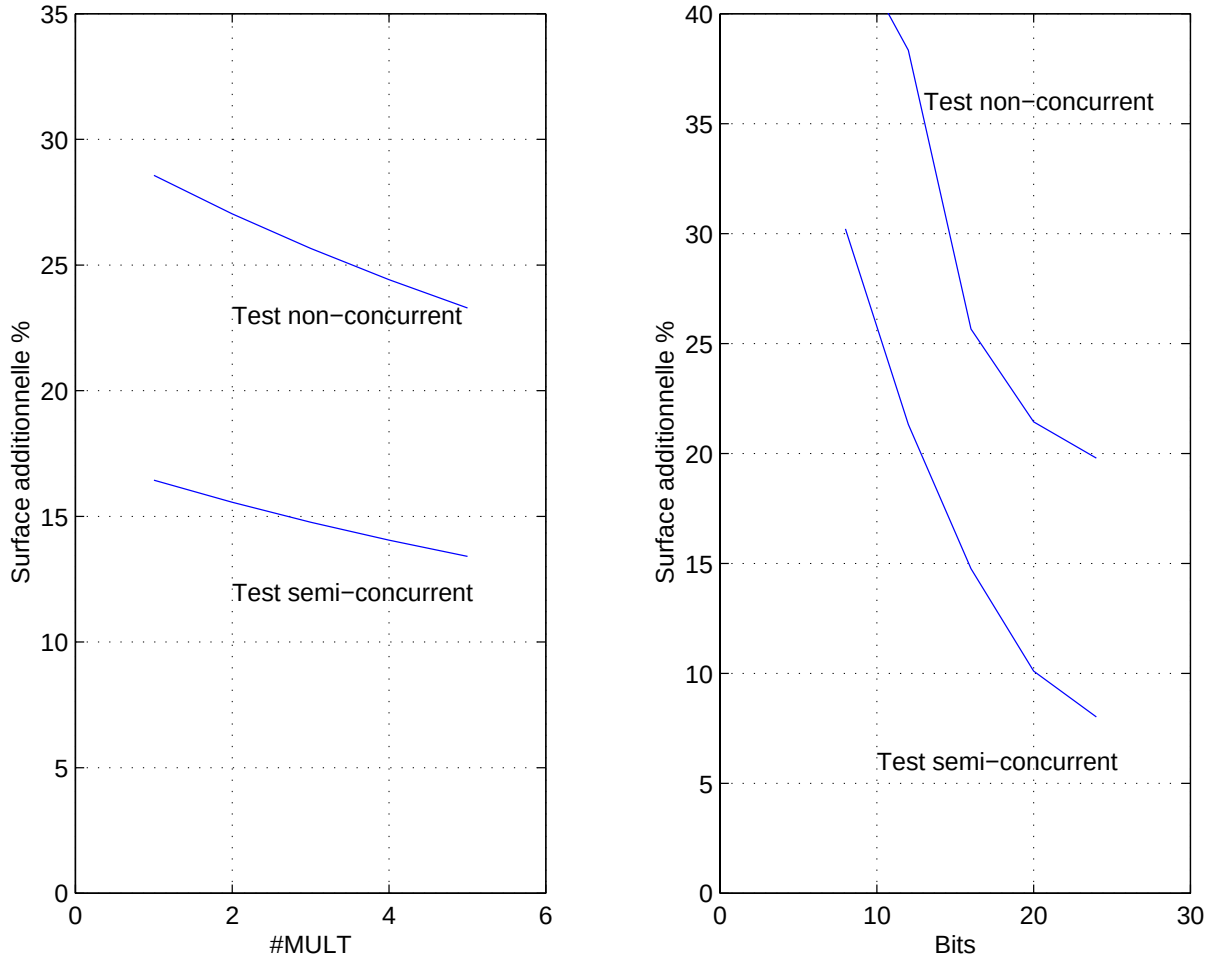


Figure 8.22: *Evaluation du coût additionnel en surface des méthodes de test en-ligne pour différentes architectures et pour différentes largeurs en bit du multiplieur (0.35 μ csx de AMS).*

Conclusion et perspectives

Dans cette étude, nous sommes partis du besoin de nouvelles solutions intégrées de test en-ligne pour les systèmes numériques complexes. Deux axes ont été développés pour répondre à ce besoin. Le premier axe considère le besoin de nouvelles solutions intégrées de test en-ligne alors que le deuxième axe adresse le problème de la synthèse de haut niveau de systèmes complexes en tenant compte des contraintes de test en-ligne mais aussi des contraintes traditionnelles de surface et de délai.

Comme solutions intégrées de test en-ligne, deux méthodes de test en-ligne ont été développées. Ces méthodes exploitent la redondance temporelle dans le système pour appliquer le test.

La première méthode, appelée méthode de test en-ligne non-concurrent, applique un test structurel sur les unités fonctionnelles. Ce test est sous forme d'un ensemble de vecteurs de test qui garantissent une couverture élevée de fautes. Nous avons utilisé l'outil design_analyzer de SYNOPSYS pour la génération de cet ensemble de vecteurs de test pour chaque type d'unités fonctionnelles.

Un circuit de test en-ligne intégré a été conçu. Son coût devient de plus en plus raisonnable en délai et en surface avec l'augmentation de la largeur en bit des unités fonctionnelles et pour les architectures parallèles. Ce circuit prend en charge la génération et l'application des vecteurs de test pendant les temps oisifs de l'unité fonctionnelle à laquelle il a été intégré. Il garantit aussi l'analyse de la réponse de l'unité fonctionnelle et la génération éventuelle d'un signal d'erreur. A noter que l'évaluation du coût en délai et en surface de ce circuit de test en-ligne s'est fait à partir d'une description comportementale de ses différents composants et en passant par une synthèse automatique par l'outil design_analyzer de SYNOPSYS. Il est important de noter que cette évaluation est à titre indicatif. Une optimisation en full-custom de certaines parties ou de la totalité du circuit peut apporter un gain en surface et en délai entre 20% et 30%.

Ce type de test en-ligne convient aux fautes permanentes qui surviennent lors du fonctionnement normal du système. Il peut être utilisé aussi pour détecter les fautes intermittentes si la durée d'affectation ou le taux d'apparition sont suffisament larges. Le système doit disposer d'un intervalle oisif suffisant pour atteindre une latence de faute déterminée.

La deuxième méthode, appelée méthode de test en-ligne semi-concurrent, applique un test fonctionnel. Ce test fonctionnel peut adresser chaque unité fonctionnelle toute seule ou l'ensemble des unités fonctionnelles du système.

Un premier test fonctionnel est appliqué par l'exploitation de la distributivité de la multiplication sur l'addition. Pour cela, les entrées fonctionnelles de l'unité sous test sont décalées (multiplication par deux) et réinjectées dans la même unité fonctionnelle, ou dans une autre

(alternance des unités fonctionnelles) si possible, pendant son temps oisif. Le résultat fonctionnel est décalé et comparé avec le résultat des entrées décalées. Un signal d'erreur est généré en cas d'inégalité. Ce type de test fonctionnel est généralisé pour comprendre l'ensemble des unités fonctionnelles du système par le test. Le circuit nécessaire pour la réalisation de ce type de test en-ligne représente un surcoût un peu élevé pour des unités fonctionnelles à une largeur en bit moins de 12 bits. Ce surcoût diminue de façon très rapide avec l'augmentation de la largeur en bit de l'unité fonctionnelle.

Un deuxième test fonctionnel est appliqué par la répétition du calcul nominal. Le résultat nominal est reproduit par un graphe de test en-ligne inséré dans les temps oisifs de l'ordonnancement nominal du système. Aucune modification n'est nécessaire pour les unités fonctionnelles. Seulement peu de registres et, éventuellement, de multiplexeurs sont ajoutés avec une modification simple de la partie contrôle. Pour ces raisons, le coût du circuit de test en-ligne de ce type de test reste raisonnable et il est dans tous les cas inférieur à 10%.

Le test en-ligne semi-concurrent convient aux systèmes qui ne disposent pas d'un intervalle oisif suffisant pour appliquer des vecteurs de test. Les entrées fonctionnelles du système sous test, utilisées pour ce type de test, sont considérées comme une longue séquence de vecteurs de test aléatoirs assurant ainsi une couverture élevée de fautes. Le circuit de test pour ce type de test en-ligne représente un coût qui devient de plus en plus raisonnable avec l'augmentation de la largeur en bit des unités fonctionnelles et pour les architectures parallèles et qui s'améliore avec les nouvelles technologies.

Le deuxième axe de cette étude représente une nouvelle méthodologie de synthèse de haut niveau pour le test en-ligne. Cette méthodologie convient aux systèmes complexes. Les problèmes de compilation et d'ordonnancement sont résolus par une combinaison d'un algorithme génétique avec une heuristique. Chaque solution, sous forme de graphe de flot de données ordonnancé, est représentée par un chromosome. Une population initiale est générée. Le voisinage des individus de cette population est exploré pour la recherche d'une solution ou pour l'optimisation d'une solution trouvée. Si aucune solution n'est pas trouvée, les opérateurs génétiques sont appliqués aux individus de la population initiale pour exploration de l'espace de solutions. Un grand soin doit être porté au réglage des paramètres de l'algorithme génétique pour réussir cette exploration.

En plus de certains détails mentionnés ci-dessus qui nécessitent une étude plus approfondie, une perspective très importante de cette étude réside dans le parallélisme intrinsèque que représente un algorithme génétique. L'exploration de l'espace de solutions peut être répartie sur plusieurs stations de travail d'un réseau local lâchement couplé. Cela ouvre la porte au développement d'un outil de synthèse de haut niveau qui tient compte des contraintes de test en-ligne et qui peut manipuler des systèmes très complexes à un coût raisonnable en temps de conception et en ressource impliquées.

Annexe A

Spécifications comportementales

La description du filtre à synthétiser peut être donnée sous trois formes : les paramètres du filtre, les coefficients et une description VHDL comportementale.

A.1 Paramètres

N : l'ordre du filtre ;

R_p **:** ripple de la bande passante ;

R_s **:** stopband decibles down ;

f_1 **:** le début de la bande passante ;

f_2 **:** la fin de la bande passante ;

f_sampling : la fréquence d'échantillonnage ;

B : le nombre de bits du codage des coefficients.

A partir de ces paramètres, les coefficients de la fonction de transfert ou ceux des équations d'état peuvent être générés en passant par matlab.

A.2 Spécifications fonctionnelles et coefficients

Un fichier matlab est utilisé pour générer un fichier text contenant les données suivantes :

The coefficients of the elliptic filter:
N= 4
Rp= 0.100000 db
Rs= 40 db
f1= 20 KHz
f2= 30 KHz
f_sampling= 500 KHz
B= 16 bits
The stat coefficients:
A=[0.681201 -0.192694 0.260985 -0.029913
0.192694 0.930851 0.029913 0.299740

-0.260985 0.029913 0.959485 0.004644
-0.029913 -0.299740 -0.004644 0.953469]
B=[0.153189
0.017558
-0.023781
-0.002726]
C=[0.107441 1.218604 0.016679 0.189173]
D=[0.019778]
The transfert function coefficients:
At=[1.000000 -3.525007 4.826040 -3.037740]
Bt=[0.019778 -0.032773 0.026067 -0.032773]

A.3 VHDL comportemental

Un fichier VHDL comportementale du filtre peut être générer :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity LDS is
        generic(n: in integer := 4;
           s: in integer := 3;
           m: in integer := 1);
port(clk,reset: in std_logic;
           u : in real;
           y : out real);
end LDS;
architecture beh of LDS is
        type real_vector is array (natural range <>) of real;
        signal x : real_vector(0 to n-1);
begin
        process(reset, u)
        begin
           if reset = '0' then
           x <= (others => 0.0);
           else
           -- x(k+1) = Ax(k) + Bu(k)
           -- y(k) = Cx(k) + Du(k)
           x(0) <= 0.681201*x(0)-0.192694*x(1)+0.260985*x(2)-0.029913*x(3)+0.153189*u;
           x(1) <= 0.192694*x(0)+0.930851*x(1)+0.029913*x(2)+0.299740*x(3)+0.017558*u;
           x(2) <= -0.260985*x(0)+0.029913*x(1)+0.959485*x(2)+0.004644*x(3)-0.023781*u;
           x(3) <= -0.029913*x(0)-0.299740*x(1)-0.004644*x(2)+0.953469*x(3)-0.002726*u;
           y <= 0.107441*x(0)+1.218604*x(1)+0.016679*x(2)+0.189173*x(3)+0.019778*u;
        end if;
        end process;
end beh;
```

Annexe B

Population initiale

Danc cette annexe, les graphes de flot de données ordonnancés qui forment la population initiale pour le filtre IIR du quatrième ordre traité dans le chapitre 8 sont présentés.

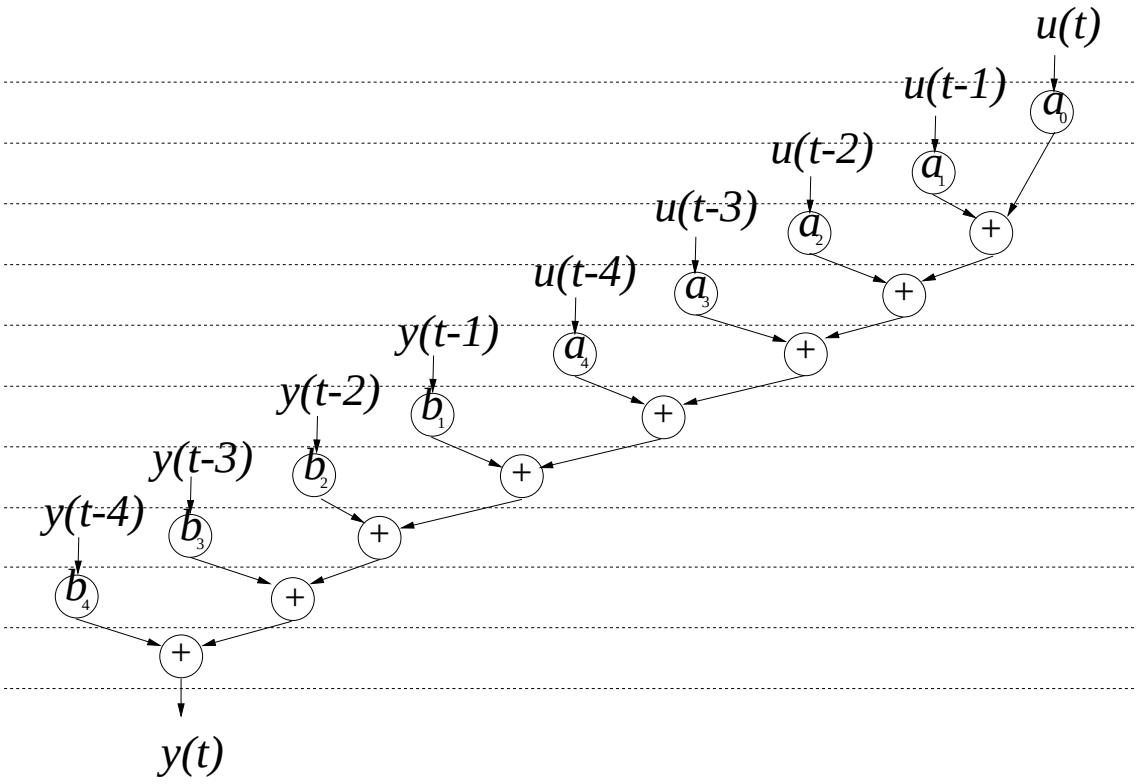


Figure B.1: *Architecture série pour forme directe 1.*

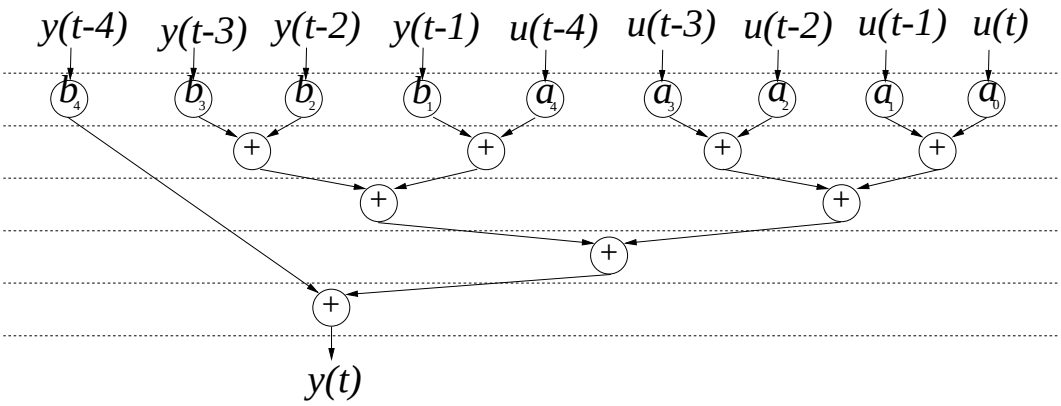


Figure B.2: *Architecture parallèle pour forme directe 1.*

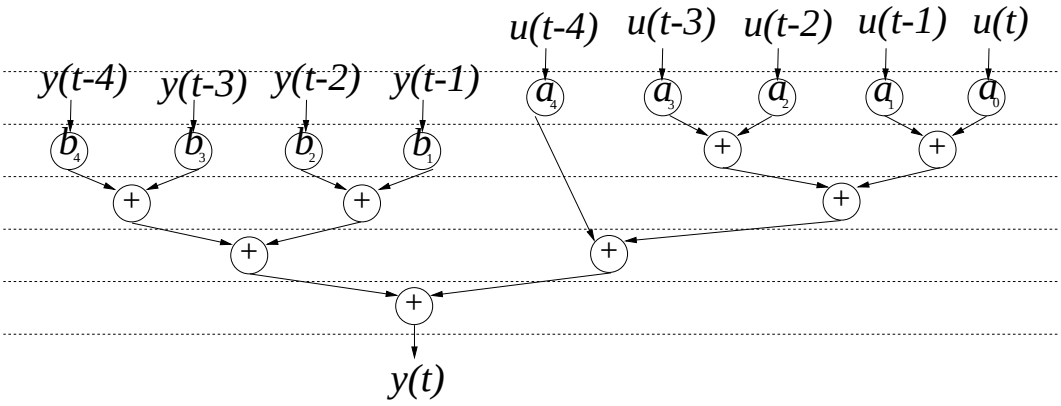


Figure B.3: *Nouvelle architecture pour forme directe 1.*

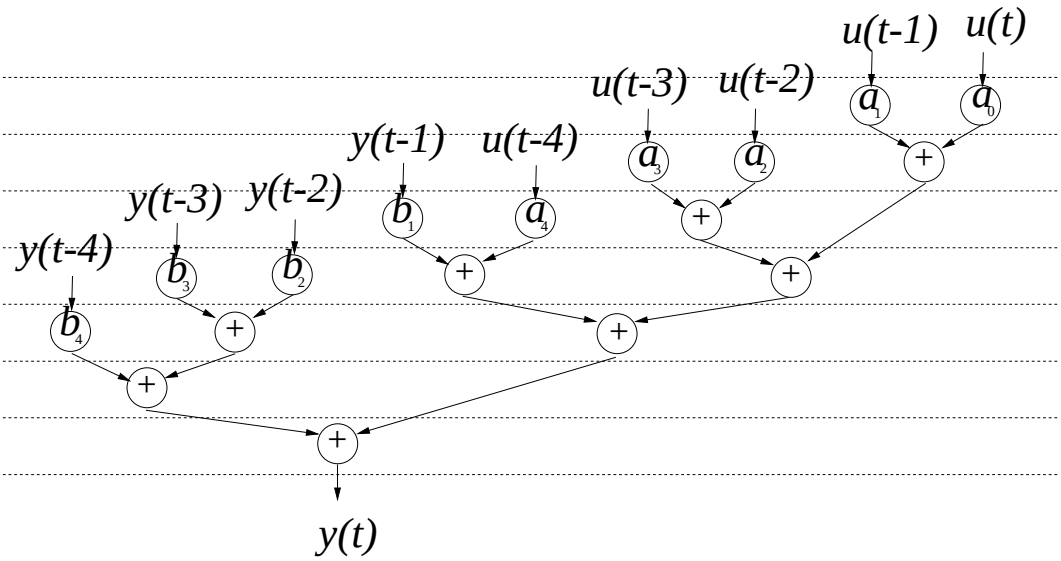


Figure B.4: *Architecture deux multiplieurs pour forme directe 1.*

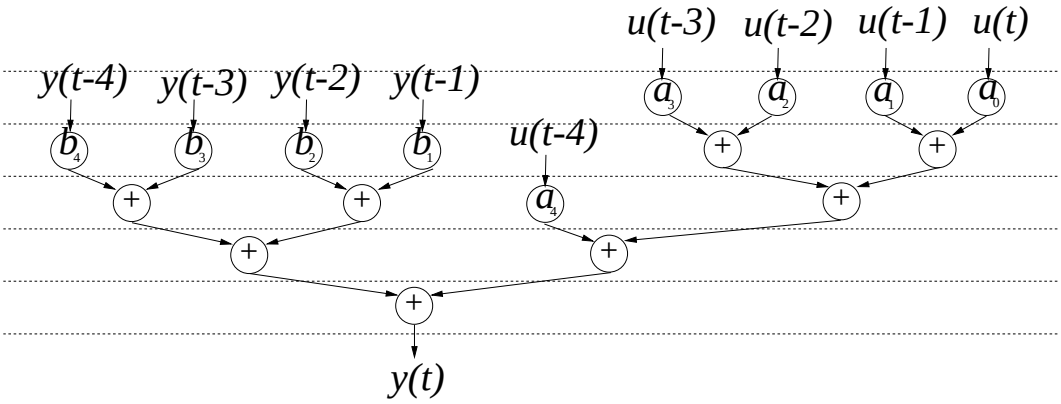


Figure B.5: *Architecture quatre multiplieurs pour forme directe 1.*

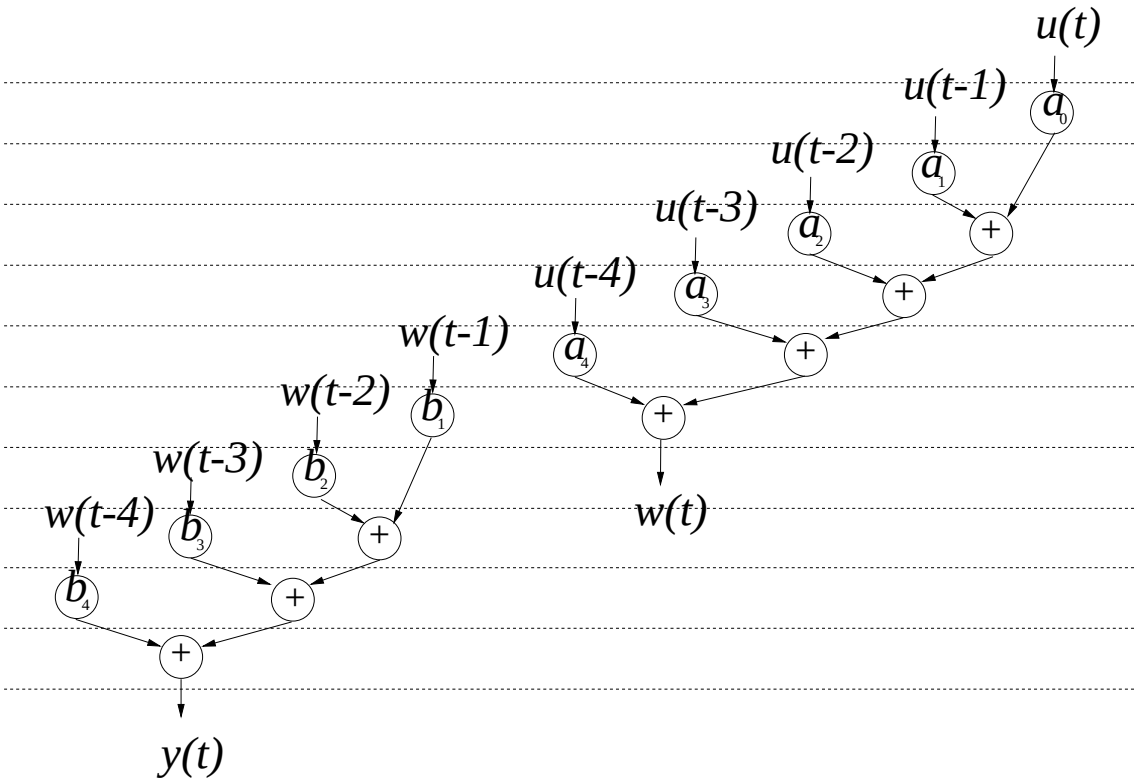


Figure B.6: *Architecture série pour forme directe 2.*

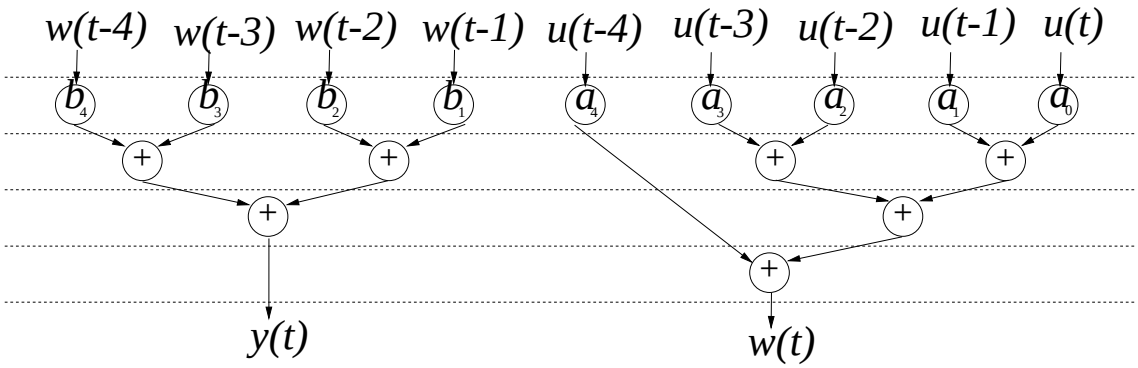


Figure B.7: *Architecture parallèle pour forme directe 2.*

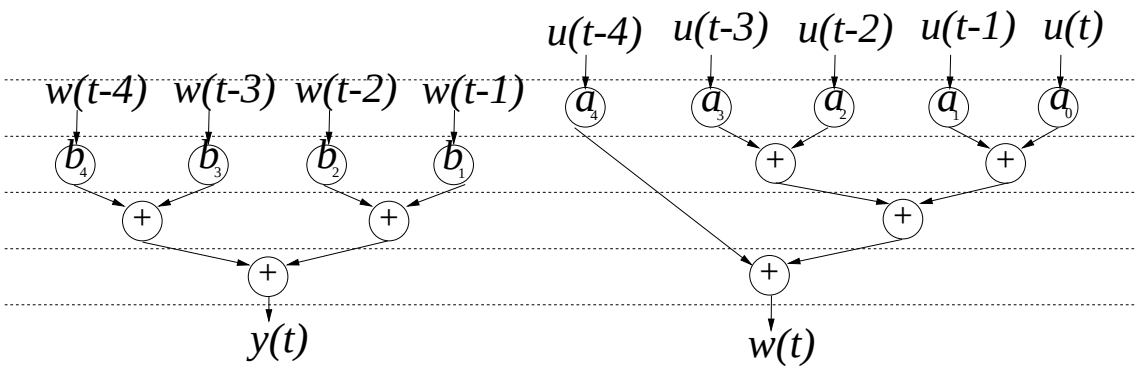


Figure B.8: *Nouvelle architecture pour forme directe 2.*

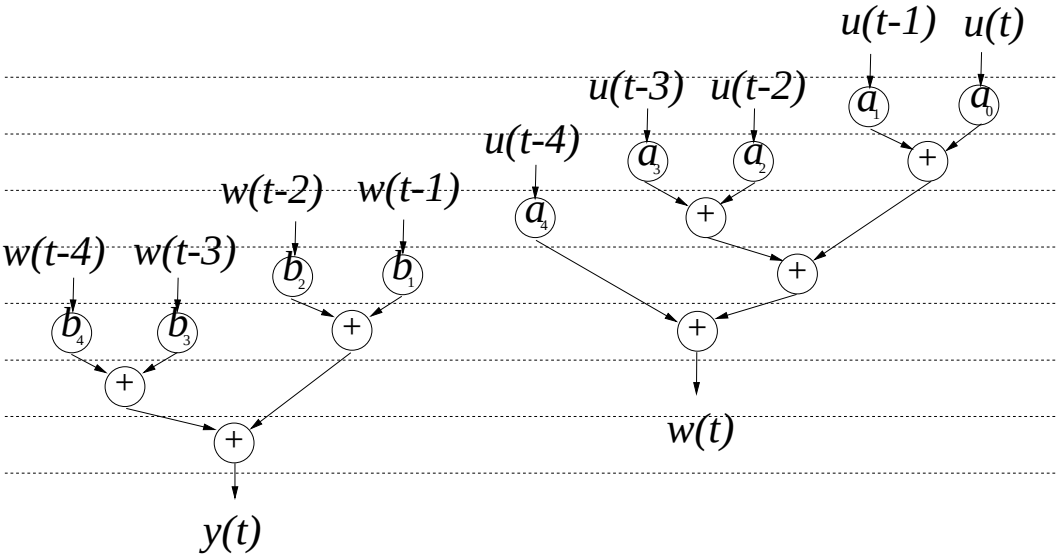


Figure B.9: *Architecture deux multiplieurs pour forme directe 2.*

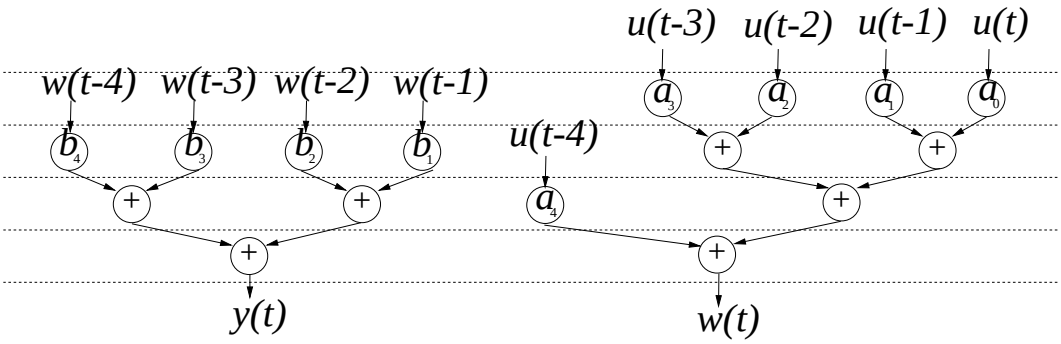


Figure B.10: *Architecture quatre multiplieurs pour forme directe 2.*

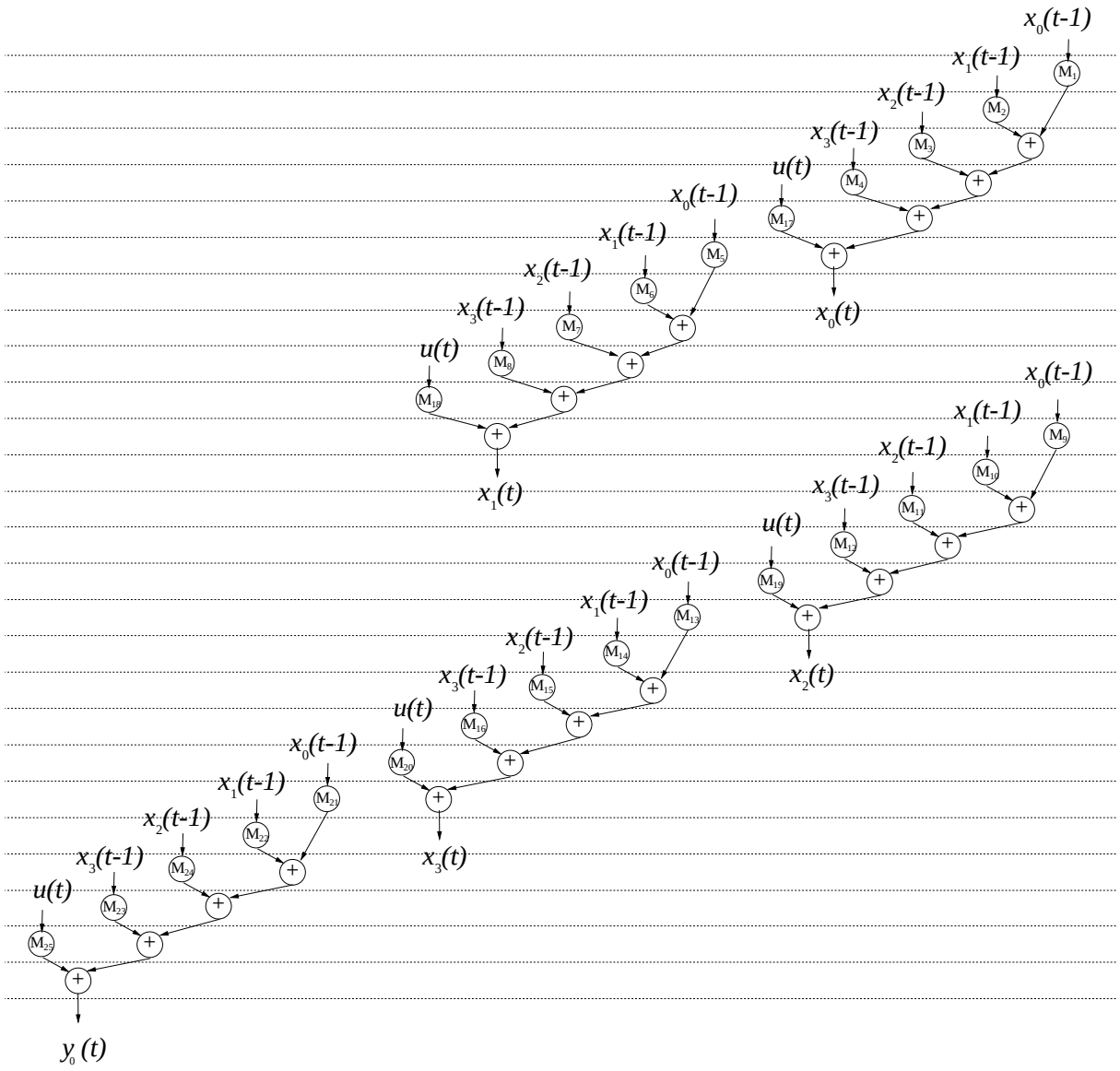


Figure B.11: *Architecture série pour équations d'état.*

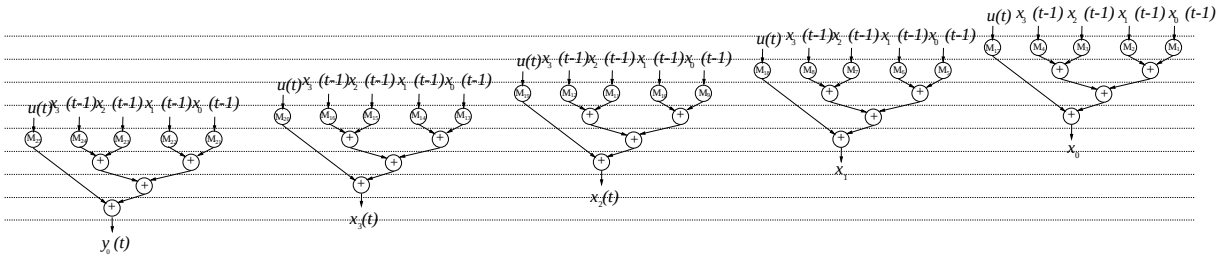


Figure B.12: *Architecture intérmédiaire et nouvelle architecture pour équations d'état.*

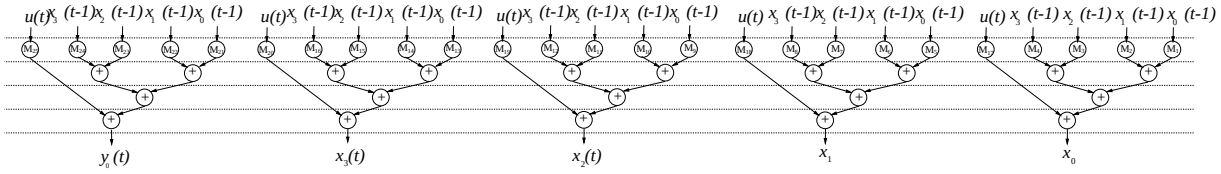


Figure B.13: *Architecture parallèle pour équations d'état.*

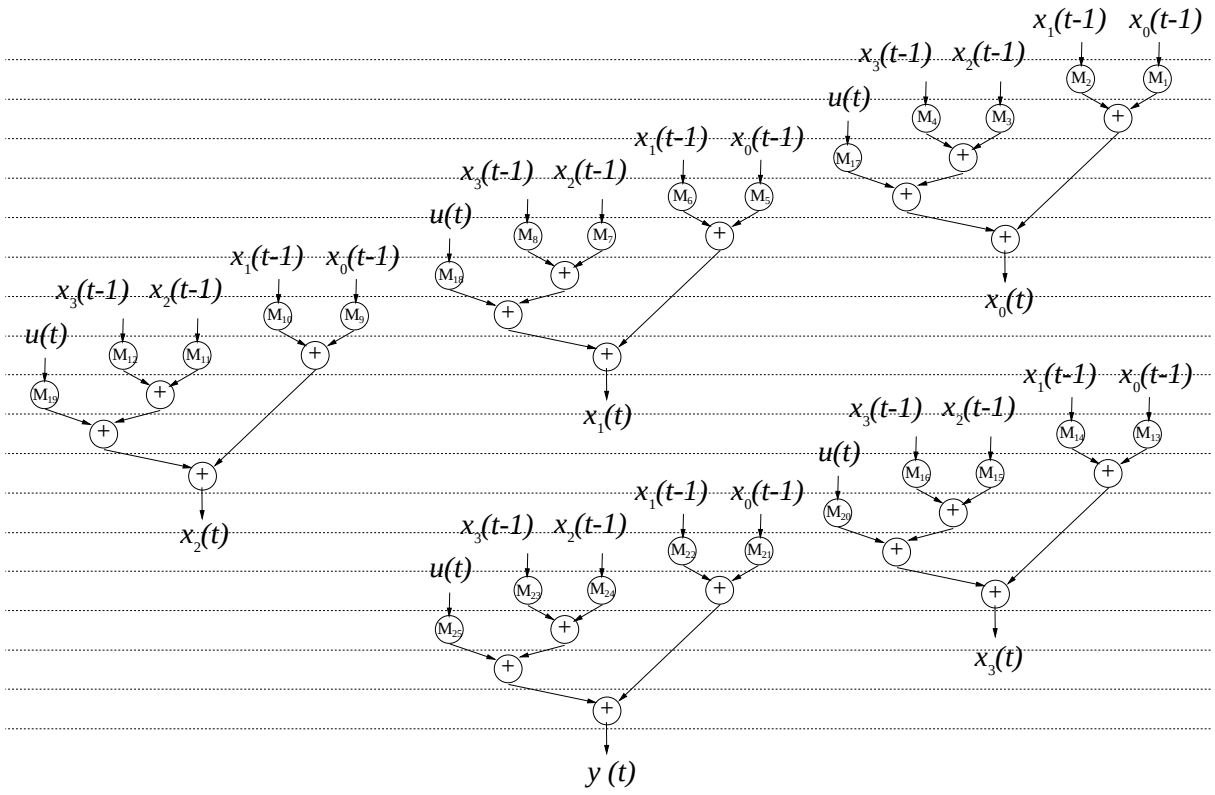


Figure B.14: *Architecture deux multiplieurs pour équations d'état.*

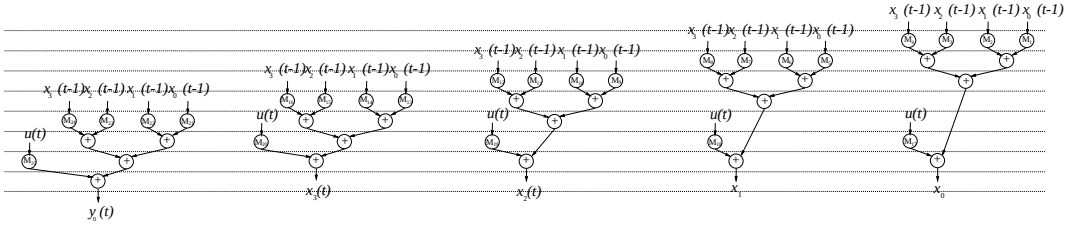


Figure B.15: *Architecture quatre multiplieurs pour équations d'état.*

Annexe C

Mécanisme de détection de faute pour le test en-ligne non-concurrent

Le mécanisme de la détection de faute est illustré par une partie de la simulation de faute. Cette partie est présentée en figure (C.1).

Avant l'injection de faute, le filtre fonctionnait de façon normale. Les signatures générées à partir de la sortie du multiplieur correspondent à celles de la réponse correcte.

A l'instant 1.2×10^5 ns une faute de type collage à 1 est injectée au bit 14 de l'entrée B du multiplieur. Les valeurs de l'entrée affectée sont modifiées.

Le temps indiqué un temps oisif du multiplieur et les vecteurs de test continuent à être appliqés sur le multiplieur.

Les vecteurs de test numéro 23 à 26 n'excitent pas la faute injectée, le circuit de test en-ligne ne signale aucune erreur. Le vecteur de test numéro 27 excite la faute et la réponse du multiplieur à l'instant suivant est erronée. La signature qui a été générée à partir de la sortie erronée ne correspond pas à celle de la sortie normale. Un signale d'erreur est généré.

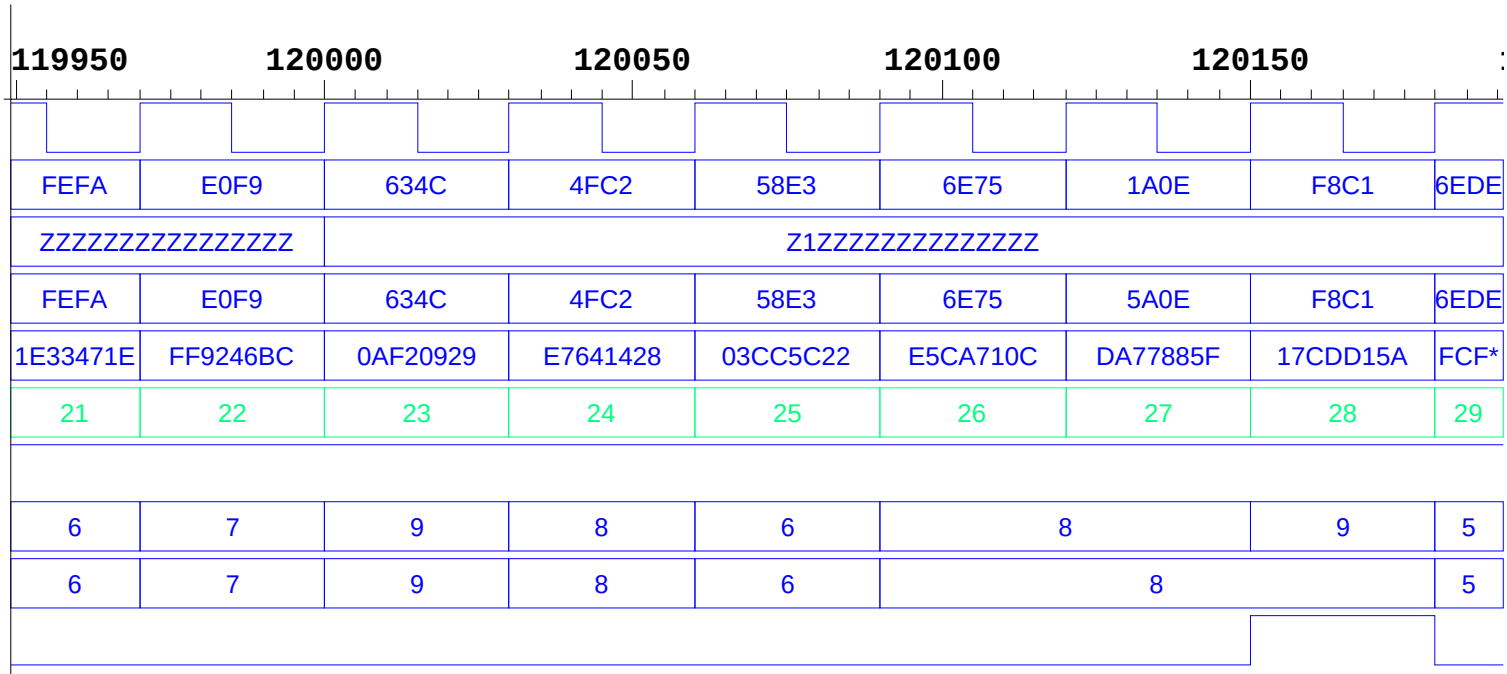


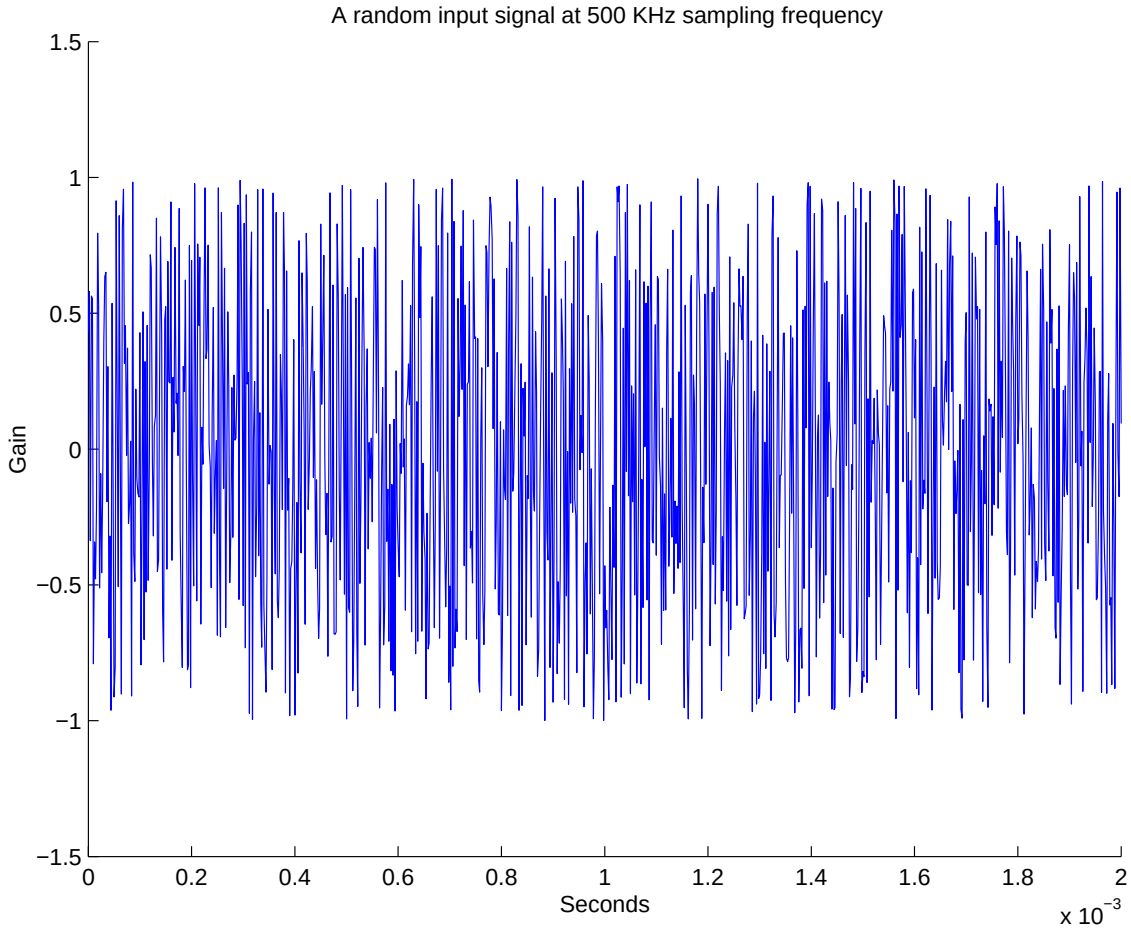
Figure C.1: *Le mécanisme de détection de faute par le test en-ligne non-concurrent.*

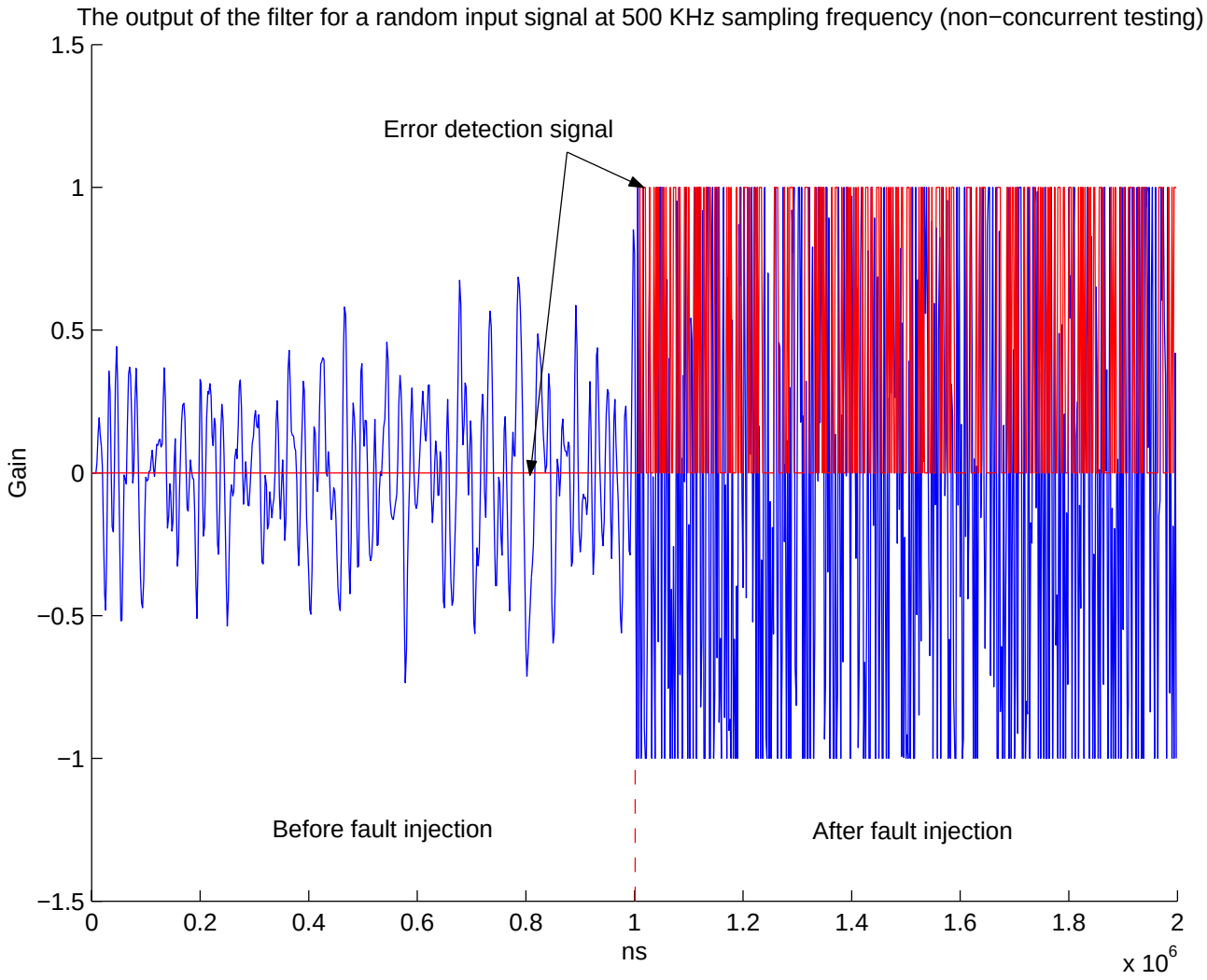
Les lignes dans cette figure représentent :

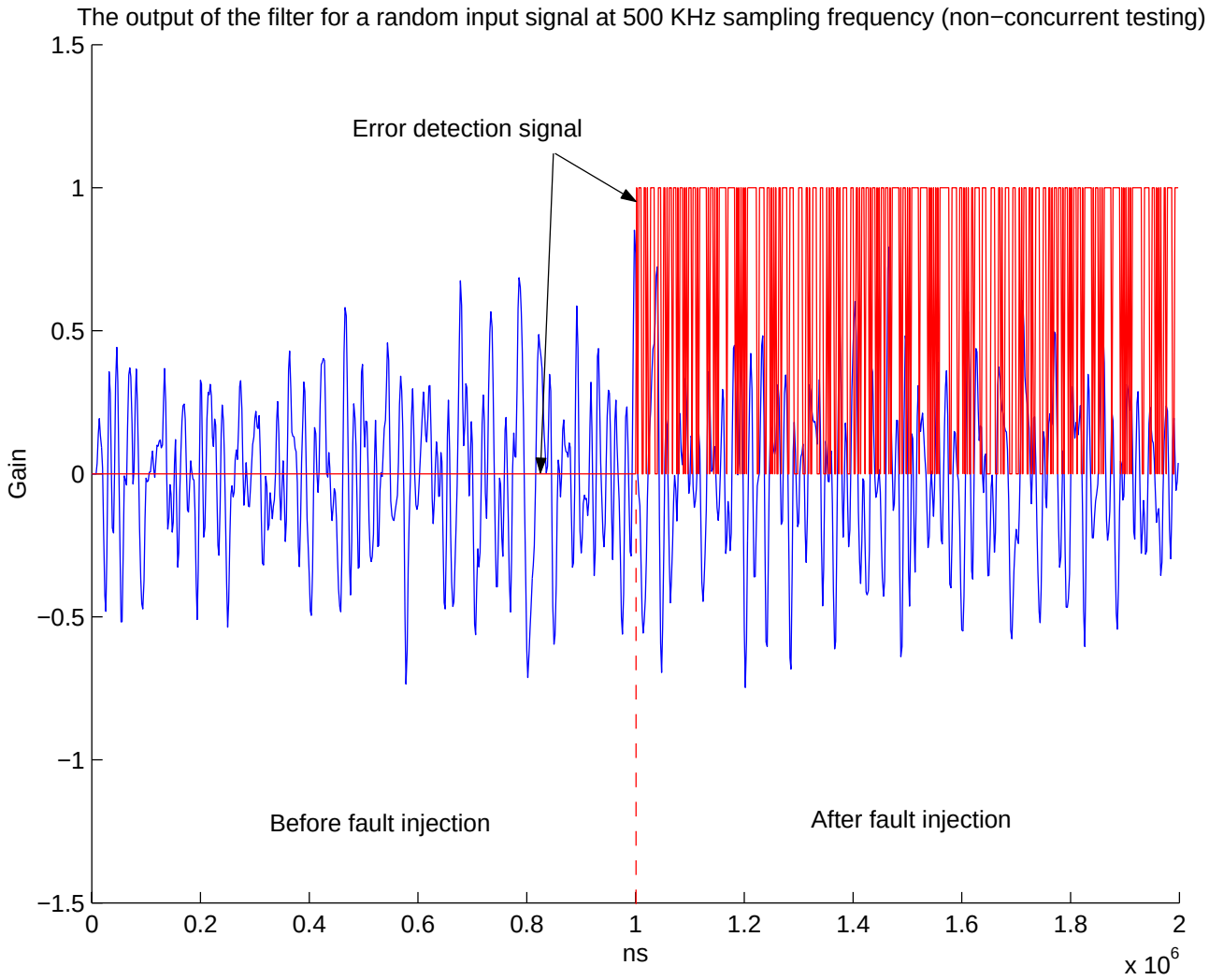
1. L'horloge fonctionnel du filtre ;
2. L'entrée du multiplieur avant son affectation par la faute ;
3. La localisation du bit affecté : Z pas de faute, 0 collage à 0, 1 collage à 1 ;
4. L'entrée du multiplieur après son affectation par la faute ;
5. La sortie du multiplieur ;
6. Numéro de vecteur de test en cours d'application ;
7. Etat oisif du multiplieur ;
8. La signature de la réponse du multiplieur ;
9. La signature correcte ;
10. Le signal de détection d'erreur.

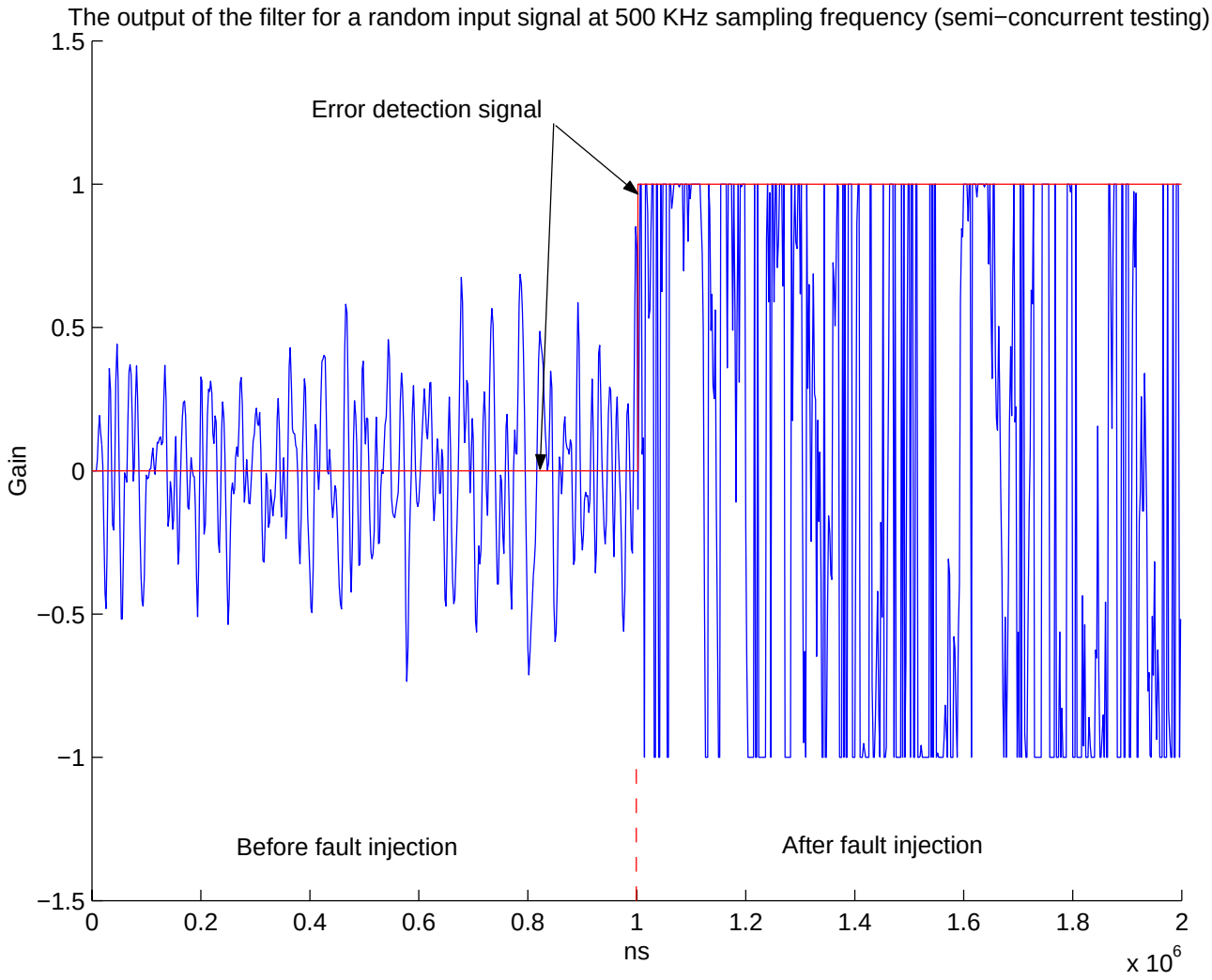
Annexe D

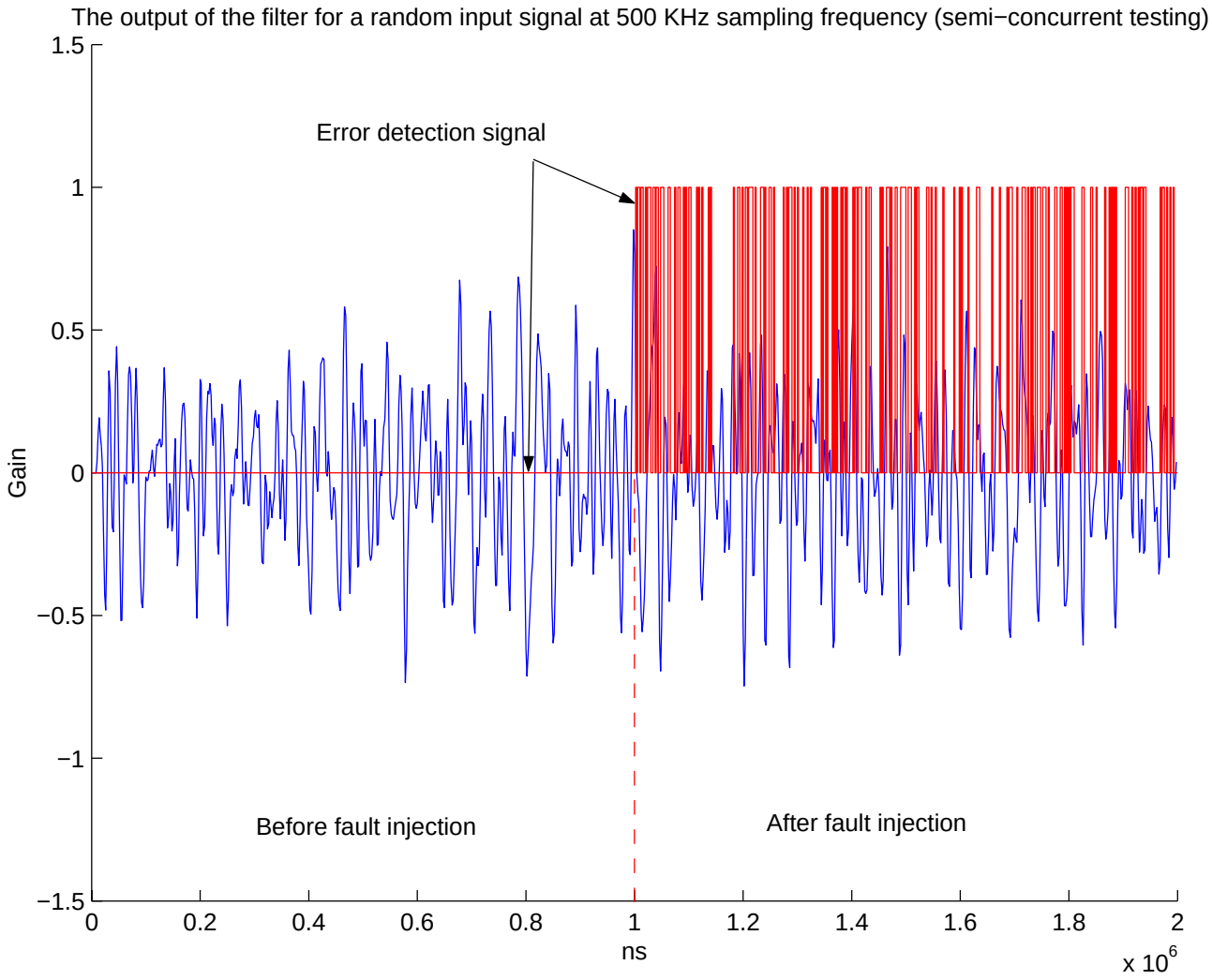
Simulation de fautes




The output of the filter for a random input signal at 500 KHz sampling frequency (non−concurrent testing)
Error detection signal
Gain
Before fault injection
After fault injection
ns
x 10^6
1.5
1
0.5
0
−0.5
−1
−1.5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


The output of the filter for a random input signal at 500 KHz sampling frequency (non−concurrent testing)
Error detection signal
Gain
Before fault injection
After fault injection
ns
x 10^6
1.5
1
0.5
0
−0.5
−1
−1.5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


The output of the filter for a random input signal at 500 KHz sampling frequency (semi−concurrent testing)
Error detection signal
Gain
Before fault injection
After fault injection
ns
x 10^6
1.5
1
0.5
0
−0.5
−1
−1.5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


The output of the filter for a random input signal at 500 KHz sampling frequency (semi−concurrent testing)
Error detection signal
Gain
Before fault injection
After fault injection
ns
x 10^6
1.5
1
0.5
0
−0.5
−1
−1.5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

Annexe E

Scripts

E.1 Le script Tcl/Tk de l'interface graphique de l'outil HLS_OLT

```
#!/usr/local/bin/wish8.2
    proc GetlogFile
    if  [winfo exists .logresults]
    raise .logresults
    else
    toplevel .logresults
    wm title .logresults "Results"
    set r .logresults
    frame $r.dum
    text $r.dum.t -setgrid true -width 80 -height 25
    -yscrollcommand .logresults.dum.s set
    scrollbar $r.dum.s -command .logresults.dum.t yview -orient vertical
    pack $r.dum.s -side right -fill y
    pack $r.dum.t -side left -fill both -expand true
    pack $r.dum
    button $r.dis -text Dismiss -command destroy .logresults
    $r.dum.t delete 1.0 end
    pack $r.dis
    wm title .logresults "The log file"
    #
    # open temp file
    #
    set tin [open log.out r] ;# don't delete earlier text
    while  [gets $tin line ] > -1
    .logresults.dum.t insert end " $line \n"
    proc Getlog  w
    set tin [open log.out r] ;# don't delete earlier text
    while  [gets $tin line ] > -1
    $w insert end " $line \n"
    proc GetAbout
```

```
if  [winfo exists .results]
raise .results
else
toplevel .results
wm title .results "Results"
set r .results
frame $r.dum
text $r.dum.t -setgrid true -width 57 -height 8
pack $r.dum.t -side left -fill both -expand true
pack $r.dum
button $r.dis -text Dismiss -command destroy .results
$r.dum.t delete 1.0 end
pack $r.dis
wm title .results "About the HLS_OLT tool"
#
# open file
#
set tin [open About r] ;# don't delete earlier text
while  [gets $tin line ] > -1
.results.dum.t insert end " $line \n"
proc Gen_RTL_bis  w
catch exec xterm -fg white -bg blue -sb -b 5 -e IIR_Tk_D_bis  msg
Getlog $w
proc Gen_FPGA  w
# catch exec xterm -fg white -bg blue -sb -b 5 -e IIR_Tk_D  msg
catch exec xterm -fg white -bg blue -sb -b 5 -e IIR_Tk_D_NZ  msg
Getlog $w
proc SetTargetLib  w
catch exec xterm -fg white -bg blue -sb -b 5 -e dc_shell -f gr_time.scr & msg
proc Gen_M  w
catch  exec xterm -fg white -bg blue -sb -b 5 -e Gen_filtgen & msg
Getlog $w
catch  exec matlab & msg
proc Gen_w  w
catch  exec Gen_waves ≫ log.out  msg
Getlog $w
proc Compile  w
catch exec vhdlan IIR_Tk.vhd ≫ log.out  msg
Getlog $w
proc cleanUp
# remove temp.tcl and destroy .results
catch exec /bin/rm temp.tcl msg
destroy .results
proc ExitHTk
# remove t.out
```

```
    catch exec rm t.out msg
    exit
    proc OpenFile
    global fileselect oldname
    fileselect
    tkwait window .fileSelectWindow
    set oldname $fileselect(selectedfile)
    set openf $fileselect(selectedfile)
    set fid [open $openf r]
    catch exec nedit $openf &
    close $fid
    source filesel.tcl
    frame .base -bg gray ;#create a frame
    wm title . "High-Level Synthesis for On-Line Testing (HLS_OLT)"
    pack .base -padx 5 -pady 5 -ipadx 2 -ipady 2 ;#pack it
    frame .menubar -relief raised -bd 2
    pack .menubar -in .base -fill x
    text .t -yscrollcommand ".sc set" ;#text widget to show log information
    scrollbar .sc -command ".t yview" ;#vertical scrollbar
    pack .sc -in .base -side right -fill y ;#make it visible
    pack .t -in .base -side left ;#make text widget visible
    menubutton .menubar.file -text File -underline 0 -menu .menubar.file.menu
    menubutton .menubar.set -text Set -underline 0 -menu .menubar.set.menu
    menubutton .menubar.generate -text Generate -underline 0 -menu .menubar.generate.menu
    menubutton .menubar.faultsim -text "Simulation" -underline 0 -menu .menubar.faultsim.menu
    menubutton .menubar.prototyping -text "Prototyping" -underline 0 -menu .menubar.prototyping.menu
    button .menubar.about -text About ... -command GetAbout;
    pack .menubar.file .menubar.set .menubar.generate .menubar.faultsim .menubar.prototyping
-side left
    pack .menubar.about -side right
    menu .menubar.file.menu
    .menubar.file.menu add command -label "Open ..." -command OpenFile
    .menubar.file.menu add command -label "Log file" -command GetlogFile
    .menubar.file.menu add command -label "VHDL filter" -command exec nedit IIR_Tk.vhd &
    .menubar.file.menu add command -label "Target library script" -command exec nedit
gr_time.scr &
    .menubar.file.menu add command -label Quit -command ExitHTk
    menu .menubar.set.menu
    .menubar.set.menu add command -label "Filter specifications" -command exec nedit -g
10x8 Filter specifications &
    .menubar.set.menu add command -label "Constraints and options" -command exec nedit
-g 10x8 Constraints and options &
    .menubar.set.menu add command -label "Target library data-base" -command SetTar-
getLib .t
    menu .menubar.generate.menu
```

.menubar.generate.menu add command -label "Filter coefficitents" -command Gen_M .t
.menubar.generate.menu add command -label "On-line testable RTL" -command Gen_RTL_bis .t
#-state disable
menu .menubar.faultsim.menu
.menubar.faultsim.menu add command -label "Filter compilation" -command Compile .t
.menubar.faultsim.menu add command -label "Fault simulation" -command exec vhdldbx &
.menubar.faultsim.menu add command -label "Set simulation results" -command Gen_w .t
.menubar.faultsim.menu add command -label "Frequency response" -command exec gv responsec.eps &
.menubar.faultsim.menu add command -label "Input signal" -command exec gv signalc.eps &
.menubar.faultsim.menu add command -label "output signal" -command exec gv IIR_waves.eps &
menu .menubar.prototyping.menu
.menubar.prototyping.menu add command -label "Generate prototype" -command Gen_FPGA .t
.menubar.prototyping.menu add command -label "Compilation" -command Compile .t
.menubar.prototyping.menu add command -label "Synthesis" -command exec xterm -fg white -bg blue -sb -b 5 -e leonardo &
.menubar.prototyping.menu add command -label "Place&Route" -command exec mm &

E.2 Le script de la génération de la base de données de la technologie cible (SYNOPSYS : design_analyzer)

link_library = "csx_HRDLIB.db "
target_library = "csx_HRDLIB.db "
symbol_library = "csxs.sdb "
default_schematic_options = "-size infinite"
Bits = 2,3,4
foreach (N, Bits)
sh "xterm -fg white -bg blue -sb -b 5 -e operator" N
read -format vhdl "/bit_est/operator.vhd"
current_design ADD
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/ADD.db" "/bit_est/ADD.db:ADD"
report_area > ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 > ./bit_est/report_t.out
set_scan_style combinational
create_test_patterns -output ADD_test_vec.vdb -compaction_effort low -check_contention true -check_float true -backtrack_effort low -random_pattern_failure_limit 64 -sample 100>tgen
write_test -input ADD_test_vec.vdb -format synopsys -output ADD_test_vec
write_test -input ADD_test_vec.vdb -format tssi_ascii -output ADD_test_vec
write_test -input ADD_test_vec.vdb -format tds -output ADD_test_vec
write_test -input ADD_test_vec.vdb -format verilog -output ADD_test_vec
write_test -input ADD_test_vec.vdb -format vhdl -output ADD_test_vec
write_test -input ADD_test_vec.vdb -format wgl -output ADD_test_vec

```
current_design MULT
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/MULT.db" "/bit_est/MULT.db:MULT"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
set_scan_style combinational
create_test_patterns -output Mult_test_vec.vdb -compaction_effort low -check_contention
true -check_float true -backtrack_effort low -random_pattern_failure_limit 64 -sample 100≫tgen
write_test -input Mult_test_vec.vdb -format synopsys -output Mult_test_vec
write_test -input Mult_test_vec.vdb -format tssi_ascii -output Mult_test_vec
write_test -input Mult_test_vec.vdb -format tds -output Mult_test_vec
write_test -input Mult_test_vec.vdb -format verilog -output Mult_test_vec
write_test -input Mult_test_vec.vdb -format vhdl -output Mult_test_vec
write_test -input Mult_test_vec.vdb -format wgl -output Mult_test_vec
current_design MUX
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/MUX.db" "/bit_est/MUX.db:MUX"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
current_design REG
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/REG.db" "/bit_est/REG.db:REG"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
current_design RCSA1
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/RCSA1.db" "/bit_est/RCSA1.db:RCSA1"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
current_design RCSA2
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/RCSA2.db" "/bit_est/RCSA2.db:RCSA2"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
current_design RCSA3
create_schematic -size infinite -gen_database
compile -map_effort medium
write -format db -hierarchy -output "/bit_est/RCSA3.db" "/bit_est/RCSA3.db:RCSA3"
report_area ≫ ./bit_est/report_a.out
report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
```

```
    current_design RCSA4
    create_schematic -size infinite -gen_database
    compile -map_effort medium
    write -format db -hierarchy -output "/bit_est/RCSA4.db" "/bit_est/RCSA4.db:RCSA4"
    report_area ≫ ./bit_est/report_a.out
    report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
    current_design RCSA5
    create_schematic -size infinite -gen_database
    compile -map_effort medium
    write -format db -hierarchy -output "/bit_est/RCSA5.db" "/bit_est/RCSA5.db:RCSA5"
    report_area ≫ ./bit_est/report_a.out
    report_timing -path full -delay max -max_paths 1 -nworst 1 ≫ ./bit_est/report_t.out
    remove_design -all
    echo N
    sh "xterm -fg white -bg blue -sb -b 5 -e report_aRCSA" N sh "xterm -fg white -bg blue
-sb -b 5 -e report_tRCSA" N sh "xterm -fg white -bg blue -sb -b 5 -e mvf" N
    quit
```

Bibliographie

[1] *http://www.tcl.tk/software/tcltk/.*

[2] A. Abdelhay. *Test en Ligne des Systèmes Digitaux Linéaires.* Ph.D thesis desertation, TIMA Laboratory, 2001.

[3] A. Abdelhay, E. Simeu, I. Sakho, and M. A. Naal. A robust fault detection scheme for concurrent testing of linear digital systems. *6th African Conference on Research in Computer Science*, Octobre 2002.

[4] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems testing & Testable Design.* AT&T Bell Laboratory and W.H. Freeman and Company, Computer Science Press. An imprint of W. H. Freeman and Company, England, 1990.

[5] V. D. Agrawal and K.-T. Cheng. Finite state machine synthesis with embedded test function. *Journal of Electronic Testing: Theory and Applications*, (1):221–228, 1990.

[6] A. Antola, F. Ferrandi, V. Piuri, and M. Sami. Semiconcurrent error detection in data paths. *IEEE Transactions on Computers*, 50(5):449–465, may 2001.

[7] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[8] P. H. Bardell. *Built-In Test for VLSI Pseudorandom techniques.* John Wiley & Sons Inc., New York, USA, 1987.

[9] M. Bellanger. *Traitement numérique du signal. Théiorie et pratique.* MASSON, Paris , France, 1987.

[10] E. Berrebi and et al. Combined control flow and data flow domonated high-level synthesis. *33rd Design Automation Conference*, pages 573–578, June 1996.

[11] S. Bhattacharya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, 21:151–166, 1999.

[12] G. Blanchet and P. Devriendt. Processeurs de traitement numérique du signal(dsp). *Techniques de l'ingenieur*, E3 565:1–30, -.

[13] S. Brown. Fpga architectural research: A syrvey. *IEEE Design & Test of Computers*, pages 9–15, Winter 1996.

[14] S. Brown and J. Rose. Fpga and cpld architectures: A tutorial. *IEEE Design & Test of Computers*, pages 42–57, Summer 1996.

[15] R. Camposano. Behavioral synthesis. *33rd Design Automation Conference*, pages 33–34, June 1996.

[16] J. E. Carletta and C. Papachristou. Behavioral testability insertion for datapath/controller circuits. *Journal of Electronic Testing: Theory and Applications*, (11):9–28, 1997.

[17] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. High-level transformation for minimizing syntactic variances. *30th Design Automation Conference*, pages 413–418, June 1993.

[18] L.-F. Chao, A. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *30th Design Automation Conference*, pages 566–572, June 1993.

[19] A. Chatterjee and R. K. Roy. An architectural transformation program for optimization of digital systems by multi-level decomposition. *30th Design Automation Conference*, pages 343–348, June 1993.

[20] C.-H. Chen and D. G. Saab. A novel behavioral testability measure. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 12(12):1960–1970, December 1993.

[21] Y. H. Choi and M. Malek. A tolerant fft processor. *Proceedings of the fault tolerant computing symposium*, pages 266–271, 1985.

[22] J. Cong and Y. Ding. On area/depth trade-off in lut-based fpga technology mapping. *30th Design Automation Conference*, pages 213–218, June 1993.

[23] A. Dammak. *Etude de Mésures de Testabilité de Systèmes Logiques*. Université de Paris-sud, Centre d'Orsay, Paris, France, 1985.

[24] R. David. *Random Testing of Digital Circuits*. Marcel Dekker Inc., France, 1998.

[25] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York 10003, 1991.

[26] S. Devadas, H.-k. T. Ma, and A. R. Newton. Redundancies and don't cares in sequential logic synthesis. *Journal of Electronic Testing: Theory and Applications*, (1):15–30, 1990.

[27] M. K. Dhodhi, I. Ahmad, and A. A. Ismaeel. High-level synthesis of data paths for easy testability. *IEE proceedings. Circuits, Devices Syst.*, 142(4):209–216, August 1995.

[28] E. Dieulesait and D. Royer. *Systèmes linéaires de commande à signaux échantillonnés*. Masson, Paris, France, 1990.

[29] R. Dorsch and H.-J. Wunderlich. Accumulator based deterministic bist. *Proceedings of the ITC 98*, pages 412–421, 1998.

[30] P. Duncan, K. Kindsfater, L. Lynette, and J. Rajeev. Stratigies for design automation of high speed digital filters. *Journal of VLSI Signal Processing*, 9:105–119, 1995.

[31] M. L. Flottes, D. Hammad, and R. B. Automatic synthesis of bisted data paths from high level specification. *European Design and Test Conference*, pages 591–598, 1994.

[32] M. L. Flottes, D. Hammad, and B. Rouzeyre. Improving testability of non-scan design during behavioral synthesis. *Journal of Electronic Testing: Theory and Applications*, (11):29–42, 1997.

[33] S. Freeman. Test generation for data-path logic: The f-path method. *IEEE Journal of solid-state circuits*, 23(2):421–427, April 1988.

[34] I. Ghosh and N. K. Jha. High-level test synthesis: a survey. *INTEGRATION, the VLSI journal*, (26):79–99, 1998.

[35] J. C. Gille and M. Clique. *Systèmes linéaires, équations d'état*. Eyrolles, France, 1990.

[36] G. Goossens and et al. Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures. *Journal of VLSI Signal Processing*, 9:49–65, 1995.

[37] L. Guerra, M. Potkonjak, and J. Rabaey. A methodology for guided behavioral-level optimization. *35th Design Automation Conference*, pages 309–314, May 1998.

[38] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[39] I. G. Harris and A. Orailoğlu. Microarchitectural synthesis of vlsi design with high test concurrency. *31th Design Automation Conference*, pages 206–211, June 1994.

[40] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms*. Wiley-Interscience, John Wiley & Sons, Inc., New York, USA, 1998.

[41] M. Haworth and W. P. Birmingham. Towards optimal system-level design. *30th Design Automation Conference*, pages 434–437, June 1993.

[42] I. Hong, D. Kirovski, and M. Potkonjak. Potential-driven statistical ordering of transformations. *37th Design Automation Conference*, pages 347–352, June 1997.

[43] S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Computers*, 5(2):113–120, February 1994.

[44] D. Houzet. Microprocesseurs. approche générale. *Techniques de l'Ingénieur*, E3 550:1–17, 1998.

[45] D. Houzet and R. J. Chevance. Microprocesseurs : Architecture et performances. *Techniques de l'Ingénieur*, E3 555:E3 1–21, 1998.

[46] S. H. Huang and et al. A tree-based scheduling algorithm for control-dominated circuits. *30th Design Automation Conference*, pages 578–582, June 1993.

[47] J. Huisken and F. Welten. Fadic: Architectural synthesis applied in ic design. *33rd Design Automation Conference*, pages 579–584, June 1996.

[48] E. C. Ifeachor and B. W. Jervis. *Digital Signal Processing A Practical Approach*. Addison-Wesley, U.S., 1993.

[49] M. Inoue and H. Fujiwara. An approach to test synthesis from higher level. *INTEGRATION, the VLSI journal*, (26):101–116, 1998.

[50] Z. Iqbal, M. Potkonjak, and A. Parker. Critical path minimization using retiming and algebraic speed-up. *30th Design Automation Conference*, pages 573–577, June 1993.

[51] A. A. Ismaeel, R. Bhatnagar, and R. Mathew. Modification of scheduled data flow graph for on-line testability. *Microelectronics Reliability*, (39):1473–1484, 1999.

[52] R. Karri and N. Mukherjee. Versatile bist: An integrated approach to on-line/off-line bist. *Proceedings of the ITC 98*, pages 910–917, 1998.

[53] R. Karri and A. Orailoğlu. High-level synthesis of fault-secure microarchitectures. *30th Design Automation Conference*, pages 429–433, June 1993.

[54] D. W. Kenneth and S. Dea. High-level synthesis for testability: A survey and perspective. *33rd Design Automation Conference*, pages 131–136, June 1996.

[55] G. Kiefer and H.-J. Wunderlich. Deterministic bist with multiple scan chains. *Proceedings of the ITC 98*, pages 1057–1064, 1998.

[56] H. B. Kim, T. Takahashi, and D. S. Ha. Test session oriented built-in self-testable data path synthesis. *Proceedings of the ITC 98*, pages 154–163, 1998.

[57] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(6):696–717, june 1992.

[58] D. M. Kwai and B. Parhami. Scalability of programmable fir digital filters. *Journal of VLSI Signal Processing*, 21(1):31–35, 1999.

[59] Y. K. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, Novemer 1997.

[60] Y.-k. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[61] S. Laha and J. H. Patel. Error detecting in arithmetic operations using time redundancy. *Proceedings of the fault tolerant computing symposium*, pages 298–305, 1983.

[62] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, c-36(1):24–35, january 1987.

[63] M. T. Lee and et al. Domaine-specific high-level modeling and synthesis for atm swith design using vhdl. *33rd Design Automation Conference*, pages 585–590, June 1996.

[64] T.-C. Lee, N. K. Jha, and W. H. Wolf. Behavioral synthesis of highly testable data path under the non-scan and partial scan environments. *30th Design Automation Conference*, pages 292–297, June 1993.

[65] T.-C. Lee, W. H. Wolf, and N. K. Jha. Behavioral synthesis for easy testability in data path scheduling. *Proceedings of the DAC 92*, pages 616–619, 1992.

[66] J. Lefermann. *Systèmes linéaires, variables d'état*. Masson, Paris, France, 1972.

[67] J. Li and R. K. Gupta. Hdl optimization using timed decision tables. *33rd Design Automation Conference*, pages 51–54, June 1996.

[68] A. Majumdar, R. Jain, and K. Saluja. Incorporating testability considerations in high-level synthesis. *Journal of Electronic Testing: Theory and Applications*, (5):43–55, 1994.

[69] A. Majumdar, R. Jain, and K. Saluja. Incorporating performance and testability constraints during binding in high-level synthesis. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 15(10):1212–1225, October 1996.

[70] S. Malik, E. M. Sentovich, and R. K. Brayton. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, January 1991.

[71] P. Mazumder and E. M. Rudnick. *Genetic Algorithms for VLSI design, Layout and test Automation*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 1999.

[72] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. IEEE*, 78(2):301–318, February 1990.

[73] M. Mehendale. Mim: Logic module independent technology mapping for design and evaluation of antifuse-based fpgas. *30th Design Automation Conference*, pages 219–223, June 1993.

[74] M. Mehendale, G. Venkatesh, and S. D. Sherlekar. Optimized code generation of multiplication-free linear transforms. *33rd Design Automation Conference*, pages 41–46, June 1996.

[75] H. C. Mike. A testability measure for hierarchical design environnements. *IEEE ETC*, pages 303–307, 1995.

[76] P. Monteiro and T. R. N. Rao. A residue checker for arithmetic and logical operations. *Proceedings of the fault tolerant computing symposium*, pages 8–13, 1972.

[77] N. Mukherjee, J. Rajski, and J. Tyszer. Testing schemes for fir filter structures. *IEEE Transactions on Computers*, 50(7):674–688, July 2001.

[78] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis for table look up programmable gate arrays. *30th Design Automation Conference*, pages 224–229, June 1993.

[79] M. A. Naal and E. Simeu. Mesure de testabilité pour la synthèse de haut niveau. *Colloque CAO de Circuits Intégrés et Systèmes*, Mai 1999.

[80] M. A. Naal and E. Simeu. On-line testability optimization in high level synthesis. *Proceedings of the 6th IEEE IOLTW00*, pages 201–206, 2000.

[81] M. A. Naal and E. Simeu. Using concurrent and semi-concurrent on-line testing during high level synthesis: an adaptable approach. *Proceedings of the 8th IEEE IOLTW02*, 2002.

[82] J. V. Neumann. Probabilistic logic synthesis or reliable organisms for unreliable components. *Automata Studies, Annals of Mathematical Studies*, (43):43–98, August 1956.

[83] A. Orailoğlu and I. G. Harris. Microarchitectural synthesis for rapid bist testing. *IEEE Transactions on Computer-Aided Design*, 16(6):573–586, June 1997.

[84] C. A. Papachristou, S. Chiu, and H. Harmanani. A data path synthesis method for self-testable designs. *28th Design Automation Conference*, pages 378–348, June 1991.

[85] K. Parhi. High-level algorithm and architecture transformations for dsp synthesis. *Journal of VLSI Signal Processing*, 9(1-2):121–143, 1995.

[86] I. Parulkar, S. K. Gupta, and M. A. Breuer. Lower bounds on test resources for scheduled data flow graphs. *33rd Design Automation Conference*, pages 143–148, June 1996.

[87] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, june 1989.

[88] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. Dsp design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, 9:23–47, 1995.

[89] J. l. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using ptolemy. *Journal of VLSI Signal Processing*, 9:7–21, 1995.

[90] D. K. Pradhan and S. M. Reddy. A design technique for synthesis of fault tolerant adders. *Proceedings of the fault tolerant computing symposium*, pages 20–23, 1972.

[91] L. R. Rabiner and B. Gold. *Theory and applicqtion of digital signal processing.* Prentice-hall, New Jeresy, USA, 1975.

[92] C. Raul. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design*, 10(1):85–93, january 1991.

[93] G. Russell and I. L. Sayers. *Advanced Simulation and Test Methodology for VLSI design.* Van Nostrand Reinhold, London, 1989.

[94] P. Sawkar and D. Thomas. Performance directed technology mapping for look_up table based fpgas. *30th Design Automation Conference*, pages 208–212, June 1993.

[95] A. Seawright and F. Brewer. High-level symbolic construction techniques for high performance sequential synthesis. *30th Design Automation Conference*, pages 424–428, June 1993.

[96] A. Sharma and R. Jain. Estimating architectural resources and performance for high-leval synthesis applications. *30th Design Automation Conference*, pages 424–428, June 1993.

[97] A. Sharma and R. Jain. Insyn: Integrated scheduling for dsp applications. *30th Design Automation Conference*, pages 349–354, June 1993.

[98] U. N. Shenoy, P. Banerjee, and A. Choudhary. A system-level synthesis algorithm with guaranteed solution quality. *Proceedings of D.A.T.E.*, March 2000.

[99] D. P. Siewiorek and E. J. McCluskey. An iterative cell switch design for hybrid redundancy. *IEEE Transactions on Computers*, C-22:290–297, August 1973.

[100] E. Simeu. *Test Aléatoire : évaluation de la testabilité des circuits combinatoires.* Ph.D thesis desertation, LAG Laboratory, 1992.

[101] E. Simeu, A. Abdelhay, and M. A. Naal. Robust concurrent self test of linear digital systems. *The 10th Anniversary Compendium of Papers from Asian Test Symposium*, 2001.

[102] E. Simeu, A. Abdelhay, and M. A. Naal. Robust concurrent self test of linear digital systems. *ATS'01 the Tenth Asian Test Symposium*, November 2001.

[103] R. Singh and J. Knight. Concurrent testing in high level synthesis. *International Conference on Computer Design*, pages 96–103, 1994.

[104] D. J. Smith. Vhdl & verilog compared & contrasted - plus modeled example written in vhdl, verilog and c. *33rd Design Automation Conference*, pages 771–776, June 1996.

[105] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, San Diego, California, USA, 1999.

[106] P. L. Soucek and T. I. Group. *Dynamic, Genetic, and Chaotic Programming.* Wiley-Interscience, John Wiley & Sons, Inc., New York, USA, 1992.

[107] J. e. a. Steensma. Testability analysis in high level data path synthesis. *Journal of Electronic Testing: Theory and Applications*, (4):43–56, 1993.

[108] A. K. Susskind. Testing by verifying walsh coefficients. *IEEE Transactions on Computers*, c-32(2):198–201, February 1983.

[109] K. Thearling and J. Abraham. An easily computed functional level testability measure. *Proceedings of the ITC 1998*, pages 381–390, 1998.

[110] M. Vahidi and A. Orailoğlu. Testability metrics for synthesis of self-testable design and effective test plans. *IEEE ETC*, pages 170–175, 1995.

[111] J. L. Van Meerbergen, P. E. R. Lippens, W. F. J. Verhaegh, and A. Van Der Werf. Phideo: High-level synthesis for high throughput applications. *Journal of VLSI Signal Processing*, 9:89–104, 1995.

[112] I. Verbauwhede and J. M. Rabaey. Synthesis for real time systems: Solutions and challenges. *Journal of VLSI Signal Processing*, 9:67–88, 1995.

[113] H. e. a. Wang. High-level synthesis of scalable architectures for iir filters using ultichip modules. *30th Design Automation Conference*, pages 336–342, June 1993.

[114] T. W. Williams and K. P. Parker. Design for testability- a survey. *Proceedings of the IEEE*, 71(1):98–112, January 1983.

[115] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transaction on parallel and distriputed systems*, 2(4):452–471, Octobre 1991.

[116] W. Wolf. Redundancy removal during high-level synthesis using scheduling don't cares. *Journal of Electronic Testing: Theory and Applications*, (11):211–225, 1997.

[117] N.-S. Woo and J. Kim. An efficient method of partitioning circuits for multiple-fpga implementation. *30th Design Automation Conference*, pages 202–207, June 1993.

[118] L. Youn Long. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2(1):2–21, January 1997.

[119] A. Y. Zomaya and S. Olariu. Special issue on parallel evolutionary computing. *Journal of Parallel and Distributed Computing*, 47(1):1–7, Novemer 1997.

Résumé

Le besoin de solutions de test en-ligne intégré est de plus en plus important. Malgré la complexité croissante de systèmes numériques, ces solutions doivent garantir un surcoût raisonnable en temps de conception, en ressources impliquées et en performance. Cela nécessite le développement de nouvelles méthodes de synthèse de haut niveau qui doivent garantir deux contraintes. La première est la possibilité de traiter des systèmes complexes à un coût raisonnable. La deuxième est la prise en compte des contraintes de test en-ligne dans les premières tâches du flot de la synthèse de haut niveau.

Pour s'accommoder à ce besoin, la présente étude propose deux axes de travail. Le premier axe consiste à proposer deux méthodes de test en-ligne, non-concurrent et semi-concurrent, présentées comme solutions intégrées (**BIST**). Le deuxième axe consiste à proposer une nouvelle méthode de synthèse de haut niveau (**HLS**) qui tient compte de la testabilité en-ligne. La prise en compte des contraintes de test en-ligne est effectuée au niveau de la compilation de la description comportementale en graphe de flot de données (**DFG**). Selon les contraintes imposées au système, une des méthodes de test en-ligne développées dans le premier axe est intégrée au système au niveau ordonnancement.

Un système numérique donné par sa description comportementale forme l'entrée de la méthode. Dans un premier temps, une optimisation orientée testabilité adresse les équations arithmétiques dans la description comportementale du système. Outre l'amélioration de la testabilité, cette optimisation peut permettre d'améliorer les performances du design final. La description optimisée est compilée en graphe de flot de données ordonnancé. La tâche de la compilation et de l'ordonnancement est résolue par une exploration de l'espace de solutions. Dans cette exploration nous introduisons le développement d'un algorithme génétique (**AG**) adapté à ce type de problèmes. Les contraintes de test en-ligne, de surface et de délai sont considérées à cette étape pour produire une solution satisfaisante. Une fois que le graphe de flot de données ordonnancé est obtenu, la méthode qui répond le mieux aux contraintes de test en-ligne est insérée dans l'ordonnancement nominal du système. L'allocation de ressource et l'assignation permettent la génération de l'architecture testable en-ligne au niveau **RTL**. La méthode a été implémentée et évaluée pour les filtres numériques récursifs **IIR**.

Mots clés : synthèse de haut niveau, compilation, ordonnancement, testabilité en-ligne, DFG, BIST, AG.

Abstract

The need of on-line testing techniques is getting more and more important. Despite the growing complexity of digital systems, the integration of such techniques in the design flow must guarantee a reasonable cost in time-to-market, implicated resource and performance of the final design. This implies the development of new high-level synthesis methods which must respect two main constraints. The first one is to handle complex digital systems in a reasonable time and resource. The second one is to take in consideration the on-line test constraints early in the high-level synthesis.

In response to this problem, the present study proposes two main issues. First, two on-line test methods, non-concurrent and semi-concurrent, are presented as integrated solutions (**BIST**). And second, a new high-level synthesis (**HLS**) method, which takes in consideration the on-line test constraints, is developed. The on-line test constraints are considered at the compilation of the behavioural specifications into a graph-based representation (**DFG**). Then one of the developed on-line test method is integrated to the design at the scheduling step. The choice of the on-line test method depends on the imposed constraints on the system.

The input of the proposed method is a behavioural specifications of a digital system. At first, an on-line testability oriented optimisation is applied on the arithmetic equations of the system. In addition of enhancing the on-line testability of the system, this optimisation can result in better design performance. The optimised behavioural description is compiled in a scheduled data flow graph (**DFG**). The compilation and scheduling tasks are resolved by an adapted genetic algorithm (**GA**). The on-line test, area and timing constraints are addressed at this stage of synthesis to produce a desirable solution. Once the scheduled **DFG** is obtained, the adapted on-line test method is inserted in the nominal scheduling of the system. Then the resource allocation and binding generate the on-line testable **RTL** structure of the system. This new high level synthesis for on-line testability method is implemented and evaluated for **IIR** digital filters.

Key words: high level synthesis, compilation, scheduling, on-line testability, DFG, BIST, GA.

ISBN 2- 84813- 000- 8 Broche
ISBN 2- 84813- 001- 6 Electronique