

ÉVALUATION DE LA FIABILITÉ D'OPÉRATEURS NUMÉRIQUES

MÉMOIRE

SOMMAIRE :

1) Présentation du laboratoire	2
a) TELECOM ParisTech	2
b) COMELEC	3
c) ARMEL	4
2) Introduction au stage	5
a) Contexte et enjeux	5
b) Problèmes posés et objectifs à atteindre	7
3) Travaux	8
a) Organisation du travail	8
b) Théorie et modèles	9
i) principe de fiabilité et taux de masquage	9
ii) masquage d'erreur	10
iii) émulation d'erreur	11
iv) application de l'émulation d'erreur	12
c) Expérimentations et réalisations	13
i) caractérisation exhaustive	13
ii) caractérisation aléatoire	15
iii) caractérisation à une erreur	17
i) caractérisation "multi cœurs"	18
d) Résultats	19
4) Conclusion	25
a) Difficultés	25
b) Perspectives	26
c) Épilogue	28
5) Annexe	29
a) Programmes	29
b) Schémas de synthèses	41
6) Références bibliographiques	49

Pour introduire ce mémoire, il peut être intéressant d'exposer quelques aspects des conditions dans lesquelles ce stage a été réalisé. Nous allons donc présenter le laboratoire d'accueil, COMELEC, en commençant par son cadre ainsi que les composantes qui le constituent. Nous évoquerons aussi le contexte et les objectifs du stage.

Nous nous pencherons ensuite sur les travaux réalisés dans l'ordre où ils ont été abordés, pour finir avec les résultats obtenus et les difficultés rencontrées.

1) Présentation du laboratoire

a) TELECOM ParisTech

TELECOM ParisTech appartient à l'Institut TELECOM mais est aussi membre du pôle ParisTech. L'Institut TELECOM, regroupe 6 grandes écoles dont les disciplines d'enseignement et de recherche sont principalement axées sur les sciences et technologies de l'information.

Le pôle ParisTech, lui, en regroupe maintenant onze, mais ses domaines sont plus vastes et couvrent pratiquement toutes les sciences de l'ingénieur. Le groupe ParisTech est lui même membre du réseau IDEA League qui rassemble cinq différents pôles de recherche européens.

Dans des approches interdisciplinaires plus applicatives, au sein de projets de recherche collaboratifs et dans la réalisation de contrats industriels, la Recherche de TELECOM ParisTech ambitionne de relever les défis de la société de l'information émergente.

Partenaire de taille de l'université Pierre et Marie Curie, entre autre au sein de l'École Doctorale d'Informatique, Télécommunications et Électronique (EDITE) de Paris, TELECOM ParisTech regroupe environ 400 chercheurs de plusieurs catégories (doctorants, enseignants, etc.)

Sa formation d'ingénieur peut être complétée par un master orienté recherche et un diplôme de doctorat ou par un master spécialisé. Il est aussi possible d'y suivre des enseignements dans le cadre de formation continue. Les différents axes de recherche à TELECOM ParisTech se répartissent sur les départements suivants :

- i. Informatique & Réseaux (INFRES)
- ii. Traitement du Signal et des Images (TSI)
- iii. Économie, Gestion, Sciences Humaines & Sociales (EGSH)
- iv. Communications & Électronique (COMELEC)

i. Informatique & Réseaux (INFRES)

La fertilisation mutuelle de l'informatique et des réseaux doit prendre en compte la dimension multiservice des systèmes pour améliorer la performance des communications de plus en plus diversifiées. La recherche du département INFRES répond à ces nouveaux défis : urbanisation numérique pour réseaux hétérogènes, modélisation et ingénierie de la connaissance, nouveaux paradigmes pour l'informatique et les réseaux du futur.

ii. Traitement du Signal et des Images (TSI)

Ce département se concentre sur l'étude de l'image sous toutes ses formes : numérique, optique... pour des applications variées : médicale, satellitaire, artistique... Mais dans ses principaux thèmes de recherche se trouvent aussi l'étude de la parole, du son ainsi que les problèmes de codage et de transmission de ces différentes informations.

iii. Économie, Gestion, Sciences Humaines & Sociales (EGSH)

Les études et les recherches que nous menons portent sur les Technologies d'Information et de Communication (TIC) selon diverses problématiques sociologiques, linguistiques, ergonomiques, cognitives, économiques, de gestion... Il s'agit principalement d'analyser les rapports qu'entretiennent les TIC.

iv. Communications & Électronique (COMELEC)

Au confluent des activités "composants", "communications", et "services", ce département regroupe l'ensemble des vecteurs technologiques de l'information multimédia. Il assure l'interface entre les derniers concepts algorithmiques et les technologies de pointe de l'optique et de l'électronique pour réaliser le transport et le traitement de l'information à l'aide des techniques les plus en amont.

b) COMELEC

Le département Communications et Électronique se répartit sur les domaines de l'enseignement, la recherche académique et la recherche sur contrats.

Ces collaborations tiennent simultanément au partage de nombreux outils et méthodes comme la simulation de systèmes, la conception assistée par ordinateur et à l'orientation vers des finalités communes, comme par exemple les systèmes de communications (radio mobile, boucle radio, fibre optique) ou le multimédia. Elles se concrétisent par de nombreuses publications communes, thèses co-encadrées, contrats de recherches communs.

L'association entre communications et électronique facilite la définition des actions de recherche, déterminantes pour la contribution au programme doctoral de l'École et pour l'appartenance de l'ensemble du département à l'Unité de Recherche associée au CNRS (UMR 5141 LTCI).

COMELEC regroupe près de 45 personnels permanents, essentiellement chercheurs ou enseignants-chercheurs, et quelques administratifs et techniciens. Il accueille en permanence près de 80 doctorants et ouvre ses laboratoires à environ 30 stagiaires chaque année.

Le département COMELEC est lui-même divisé en huit groupes de recherche :

- Systèmes et Électroniques Numériques (SEN) ;
- Systèmes Intégrés Analogiques et Mixtes (SIAM) ;
- Communications Numériques (COMNUM) ;
- Optoélectronique et Communications Optiques (GTO) ;
- Radiofréquences et Micro-ondes (RFM) ;
- "Exploration d'Architectures de Systèmes Intégrés" (LabSoc) ;
- SMART qui réunit les administratifs et techniciens du département (SMART).

Le travail proposé au sein de ce stage s'inscrit dans le cadre d'une coopération entre plusieurs de ces groupes. L'équipe d'accueil est principalement répartie entre les groupes SEN et SIAM, nous allons donc évoquer leurs thématiques de travail.

Les activités de recherche du groupe SIAM sont aujourd'hui structurées autour de deux axes principaux : Radio logicielle (interface radio reconfigurable) et Nanotechnologies (architectures et circuits nanoélectroniques)

L'équipe SIAM étudie et propose des solutions d'intégration en technologie CMOS d'interfaces reconfigurables radiofréquences - bande de base mais également des interfaces d'acquisition reconfigurables pour capteurs. Par le biais de collaborations avec les grandes entreprises du secteur des semi-conducteurs ou de domaines applicatifs, le groupe SIAM peut valider ses méthodes, algorithmes et architectures par la réalisation et le test de circuits intégrés dans les technologies les plus récentes.

La recherche de l'équipe SEN, elle, gravite autour des architectures des systèmes embarqués communicants sécurisés. Elle est orientée par deux thèmes principaux : d'une part , les architectures électroniques des systèmes embarqués en vue de leur intégration efficace sur puce ou sur carte ; d'autre part la sécurisation des circuits contre les attaques par canaux cachés (attaques en analyse de l'activité du circuit ou en injection de faute).

Le sujet de ce mémoire étant une conséquence de la coopération entre ces groupes SIAM et SEN nous allons maintenant aborder la structure unificatrice, en présentant le projet de recherche ARMEL.

c) ARMEL (ARchitecture et Méthodes pour l'ELectronique)

Le projet ARMEL fédère des recherches en électronique numérique, analogique et mixte portant sur des méthodes d'implantation et la conception de systèmes optimisés de traitement matériel de l'information. Son objectif est de lever les verrous existants pour l'implantation de tels systèmes, qu'ils soient liés à la complexité de la fonction à mettre en œuvre ou aux propriétés du support physique considéré. Ce projet se déroule autour de deux thèmes principaux, à savoir le thème systèmes complexes et le thème nouvelles technologies.

Dans le thème systèmes complexes, les travaux sont tournés vers la gestion de la complexité algorithmique et la recherche de nouvelles solutions architecturales numériques, analogiques ou mixtes permettant l'implantation optimisée de fonctions nouvelles (nouveaux algorithmes pour de nouvelles applications) ou caractérisées par l'importance et la sévérité de leurs contraintes (précision, bande passante, puissance de calcul, surface, consommation).

Dans le thème nouvelles technologies, les travaux sont tournés vers la mise en place de nouvelles stratégies (méthodes, outils) rendant possible, rapide, fiable et viable économiquement la conception à base de technologies nouvelles (nano-technologies ou CMOS "ultime"). Ceci requiert des modèles adaptés des composants de base, de nouveaux circuits pour les fonctions élémentaires et des architectures robustes élaborées à partir de ces nouveaux circuits.

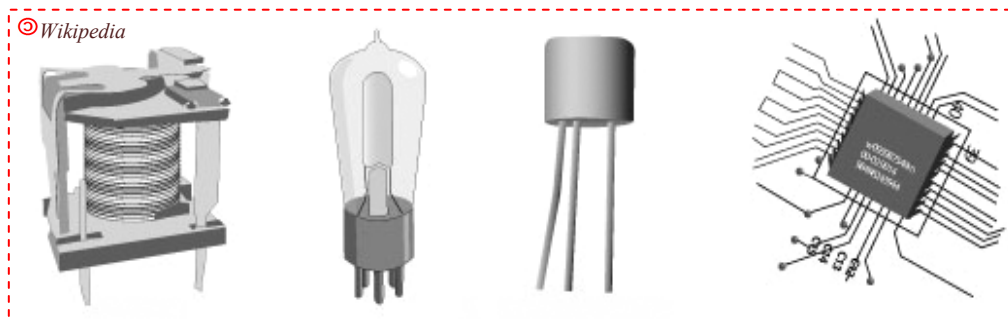
Un travail de thèse, inscrit dans le thème nouvelles technologies, est actuellement sur le point de se finir. Ce travail porte sur la fiabilité des opérateurs numériques tout comme ce mémoire de stage puisqu'il s'agit d'une contribution à cette thèse.

2) Introduction au stage

a) Contexte et enjeux

Le travail proposé dans ce stage s'inscrit donc dans le cadre d'un projet de recherche du département COMELEC portant sur les architectures du futur. Ce projet a pour objectif de contribuer à la mise au point de méthodes de conception de processeurs fiables.

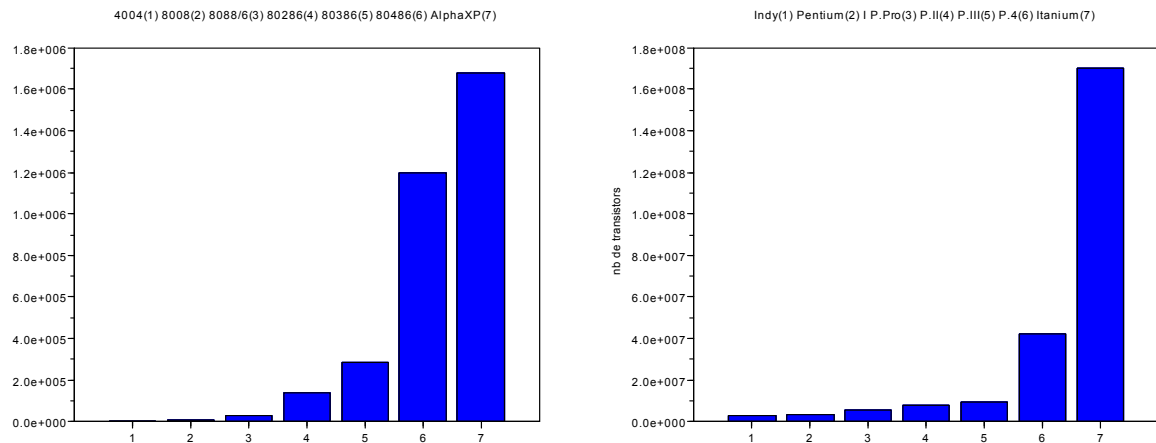
En effet, l'évolution régulière des finesses de gravure en microélectronique joue un rôle fondamental dans le développement économique depuis une cinquantaine d'années. Cependant, les technologies nanométriques conduisent à des rendements de fabrication nettement inférieurs à ceux obtenus avec les générations précédentes du fait de la proximité des limites physiques. Cette baisse de rendement et une variabilité paramétrique fortement accrue entraînent une moindre fiabilité des systèmes conçus.



L'arrivée aux dimensions nanométriques pose donc des problèmes de plus en plus complexes. Parmi les problèmes anticipés, la fiabilité d'opération a déjà montré ses effets et complique sérieusement l'emploi des nouvelles technologies. Cette menace oblige à un changement du flot traditionnel de projet des systèmes intégrés, en considérant la fiabilité comme une contrainte de conception dès le début du développement.

La réduction constante des dimensions des structures intégrées a permis à la technologie CMOS d'évoluer à un taux prévu par la loi de Moore [4]. Les gains en vitesse, surface et consommation étaient une conséquence directe de la réduction d'échelle des composants. Lorsque les dimensions des composants intégrés s'approchent du nanomètre, cette évolution n'est plus simple, et la réduction des circuits pose des problèmes qui empêchent des gains directs observés dans le passé. Comportements quantiques, courants de fuite, bruits thermiques et fluctuations paramétriques sont quelques exemples des difficultés qui se présentent de façon non négligeable à l'échelle nanométrique.

Évolution des finesses de gravure : nombre de transistors par puces © Fabien Celaia – developpez.com



4004	8008	8088/6	80286	80386	80486	Alpha XP	Indy	P. I	P. Pro	P. II	P. III	P. IV	Itanium
2250	3500	29000	136000	285000	1200000	1680000	2600000	3100000	5500000	7500000	9500000	42 E+9	170 E+9

En considérant des fluctuations paramétriques et les dimensions qui dépassent la précision lithographique, le rendement de fabrication des circuits CMOS sera de plus en plus réduit. Les architectures reconfigurables sont une solution d'ailleurs particulièrement étudiée pour une conception ascendante de circuits intégrés [8], [9].

En considérant le bruit thermique, la réduction de la tension d'opération et l'augmentation de fréquence, les circuits intégrés seront de plus en plus vulnérables aux fautes transitoires et l'utilisation de redondance spatiale est une des méthodes proposées pour augmenter la fiabilité des circuits CMOS nanométriques [5], [6], [7].

Dans le cadre de technologies à un faible rendement et de circuits plus vulnérables, ce projet de recherche propose donc d'étudier l'évaluation de la fiabilité d'opérateurs numériques afin de développer des architectures robustes pour l'implémentation de systèmes de traitement du signal.

b) Problèmes posés et objectifs à atteindre

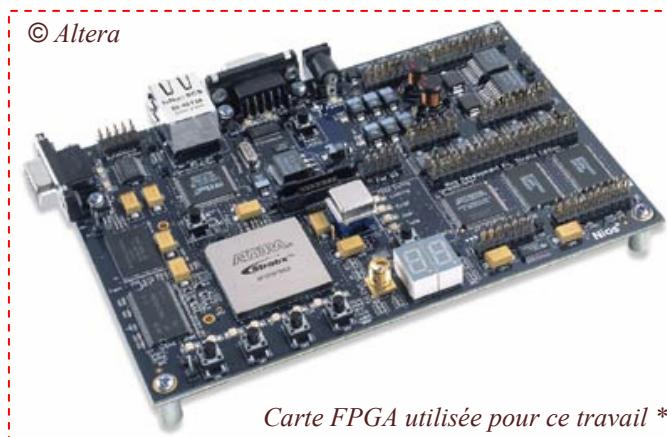
De nombreuses architectures d'opérateurs robustes ont déjà été proposées dans la littérature scientifique. C'est sur l'étude comparative de ces solutions architecturales que portera le travail proposé. Les objectifs du stage sont de contribuer à mettre en oeuvre un outil d'évaluation de la robustesse puis de participer à l'étude des compromis robustesse-coût-vitesse d'opérateurs numériques.

Plusieurs méthodes de caractérisation de circuits numériques existent. Les travaux sur les PTM (Probabilistic Transfer Matrix [1]) ont longtemps été une des références en la matière. Le principe de cette méthode est donc de formaliser le calcul de la probabilité d'erreur d'un circuit entier en fonction de la probabilité d'erreur de chaque porte logique.

Cette méthode est monstrueusement coûteuse en temps de calcul et utilisation de mémoire. Une autre solution a été proposée pour sa réalisabilité, le modèle de fiabilité PBR, Probability Binomial Reliability [2], développé au laboratoire COMELEC.

La méthode PBR, qui sera expliquée au chapitre 3) de ce document, est bien moins coûteuse, mais elle reste irréalisable par simulation dans le cas de caractérisations d'opérateurs particulièrement massifs. Par exemple, il n'est pas encore possible de caractériser un opérateur numérique 8 bits en moins d'une dizaine d'heures de simulation.

Le but de ce stage a donc été de mettre en oeuvre une accélération conséquente de cette méthode de caractérisation grâce à l'utilisation de FPGA. En effet, il est vraiment intéressant de profiter du parallélisme offert par le "câblage" de notre plateforme de caractérisation. On peut effectuer un calcul à chaque cycle d'horloge de FPGA au lieu de plusieurs cycles pour n'avoir la valeur que d'un seul nœud de circuit dans le cas d'une simulation classique.



On verra par exemple qu'on peut très facilement accélérer 60 fois le temps de caractérisation d'un opérateur 8bits (14 minutes de compilation et calculs au lieu de 14h de simulation).

Ce procédé a permis donc dans un premier temps de caractériser une bibliothèque d'opérateurs développés au laboratoire d'accueil.

Dans un avenir proche, il serait envisageable de développer un procédé réduit, mais représentatif, permettant une caractérisation en temps réel pour une intégration dans le processus de synthèse.

Le stage s'est donc appuyé sur des travaux déjà développés par l'équipe : une bibliothèque d'opérateurs numériques, un modèle de fiabilité et une méthode d'injection de fautes.

Le travail à accomplir fut donc réparti sur plusieurs étapes :

- Étudier les publications de l'équipe et assimiler les connaissances requises ;
- Examiner les opérateurs de la bibliothèque développée par l'équipe ;
- Mettre en oeuvre la méthode d'injection de fautes développée ;
- Évaluer la robustesse des opérateurs de la bibliothèque.

* Nios Development Board, Stratix II Edition (EP2S60F672C5)

3) Travaux

a) Organisation du travail

Pour mener à bien ce stage, un plan de travail a préalablement été établi. Après avoir assimilé les notions théoriques indispensables et avant d'appliquer ces méthodes découvertes, le premier besoin était de mettre en place un système de communication FPGA - ordinateur.

En effet, nous allions effectuer des calculs longs sur FPGA, le rapatriement des résultats, impliquaient la mise en place de cette communication.

Plusieurs solutions étaient envisageables, la moins coûteuse en surface aurait certainement été d'implémenter un système utilisant le protocole UART. Cette solution aurait été trop longue à développer car il aurait aussi fallu mettre en place le système logiciel de réception des données.

Il nous a donc paru plus judicieux d'adopter une démarche de CoDesign, c'est-à-dire utiliser un micro processeur décrit en langage VHDL et le programmer à l'aide de langage C embarqué.

La solution qui a été sélectionnée présente plusieurs atouts :

- homogénéité entre matériel et logiciel (interfaces et mapping logiciels générés par les outils de développement) ;
- communication UART pré-établie, on a utilisé dans notre cas l'interface JTAG, connectée en USB, normalement faite pour le debug logiciel
- téléchargement matériel et logiciel, ainsi que le debug logiciel, avec ce même lien JTAG

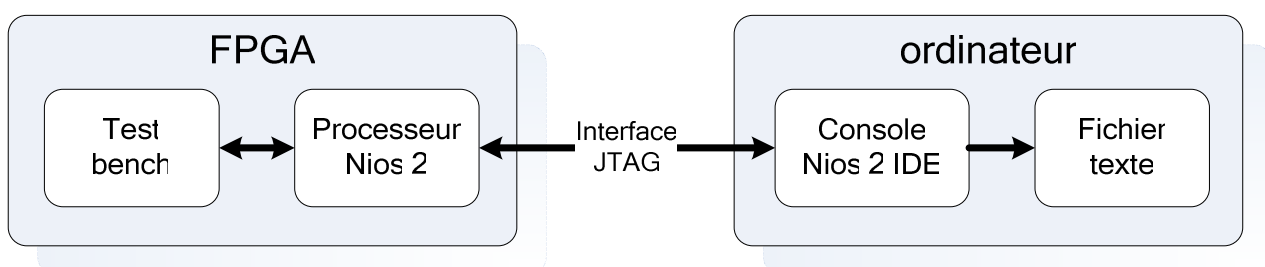
Nous avons donc utilisé le Nios2, un Processeur softcore développé par Altera. L'implémentation du Nios2 ainsi que de la partie "hard" de la caractérisation s'est faite à partir de Quartus. Bien que cette chaîne de développement logiciel, et les outils de synthèse, placement & routage FPGA (Quartus) soient tout à fait standard, des outils spécifiques en amont et en aval sont proposés :

- le développement du coeur de processeur et de ses composants ("Briques IP") s'effectue à l'aide du logiciel "SOPC Builder".
- le développement du logiciel embarqué s'effectue sous Nios2 IDE en C, la compilation des bibliothèques est faite en fonction des périphériques et options du processeur.

On notera que Nios2IDE contient aussi l'interface de debug logiciel, ce qui nous a permis de rapatrier les données calculées par affichage sur *console* et écriture dans un fichier. Cette interface est une des principales simplifications qu'offre ce choix de conception. On pourra trouver en

Annexe b la description matérielle ainsi que le programme utilisé pour ce travail.

Voici le schéma de *communication* avec la plateforme de caractérisation (test bench) :



Après cette tâche accomplie, il a été utile de se consacrer entièrement au sujet du stage. Mais avant d'arborer ce travail, il est important de présenter le principe de mesure de fiabilité d'opérateurs numérique.

b) Théorie et modèles

i) principe de fiabilité et taux de masquage

Pour mesurer la fiabilité d'opérateurs numériques, nous utiliserons une méthode basée sur un modèle développé à COMELEC : *Probabilistic Binomial Reliability* [2].

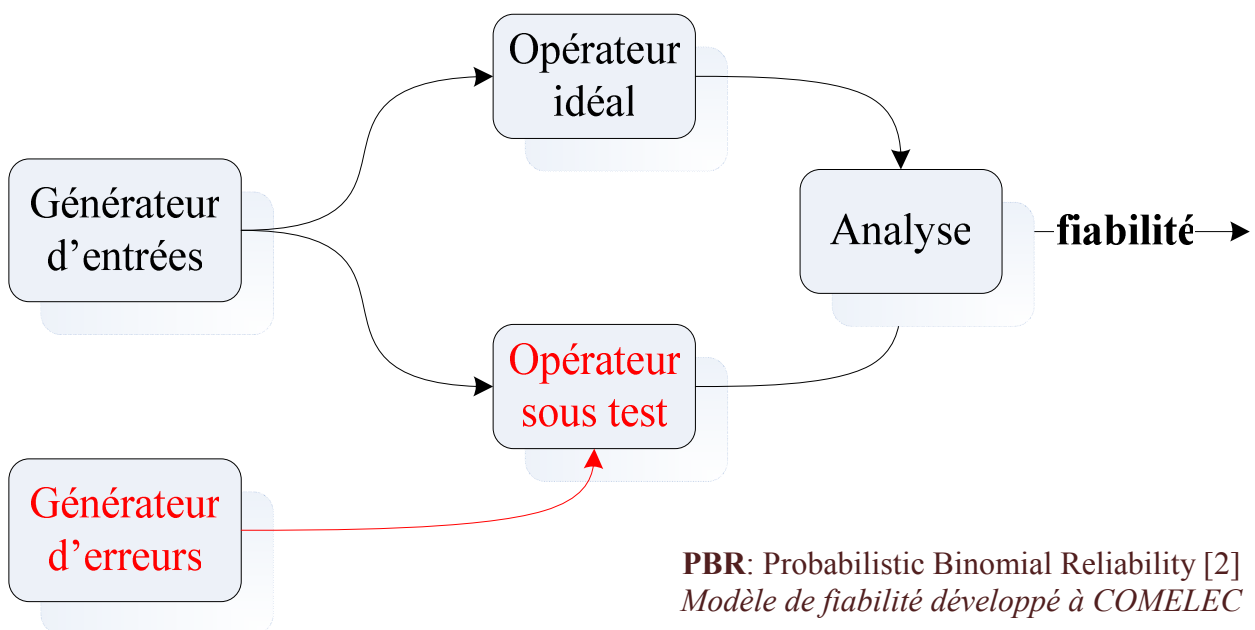
Pour présenter cette méthode, nous allons la décrire d'un point de vue global et approfondir les explications au fur et à mesure.

Le terme fiabilité que nous utilisons désigne plus précisément la *fiabilité de signal de sortie* d'un composant électronique combinatoire. Elle correspond à la probabilité que la sortie de ce composant soit correcte. Pour la déterminer, il est indispensable de connaître la probabilité d'erreur d'une porte logique, mais le taux de masquage d'erreur est tout autant nécessaire.

Pour mesurer le taux de masquage d'erreur nous allons émuler des fautes volontairement injectées à l'intérieur du circuit testé. On obtient ce taux en calculant le pourcentage de cas, où malgré ces injections d'erreurs, la sortie est correcte.

Cette méthode s'applique autant à des fautes matérielles de fabrication que des fautes passagères dues à des radiations.

Voici le schéma de principe de ce dénombrement :



Avant de pousser les explications de cette méthode, il peut être intéressant de rappeler brièvement ce qu'est un masquage d'erreur.

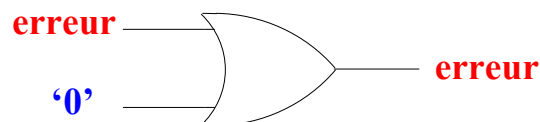
ii) masquage d'erreur

Pour expliquer ce qu'est un masquage d'erreur nous allons prendre un exemple simple, le cas de la porte logique OR :

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

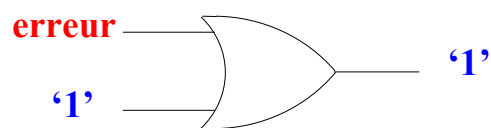
Si l'on étudie la table de vérité de ce composant, on peut constater que le fait de mettre une entrée (a) au niveau logique '0' ne modifie pas l'effet de l'autre entrée (b) sur la sortie (on désigne le niveau logique '0' comme élément neutre de cette opération).

La présence d'une erreur en (b) serait donc *recopiée en sortie* :



Si par contre on met une entrée (a) au niveau logique '1' alors, on *force* la sortie à ce même niveau logique '1'.

La présence d'une erreur en (b) n'aurait donc *aucun effet en sortie* :



=> Ce phénomène, le fait que cette erreur n'ait parfois aucun effet en sortie, est donc appelé *masquage d'erreur*.

Pour mesurer le taux de *masquage d'erreur* nous allons donc émuler des fautes volontairement injectées à des nœuds particuliers du circuit étudié.

Avant de développer le choix des nœuds particuliers, nous allons maintenant rappeler comment nous émuloons une injection d'erreur.

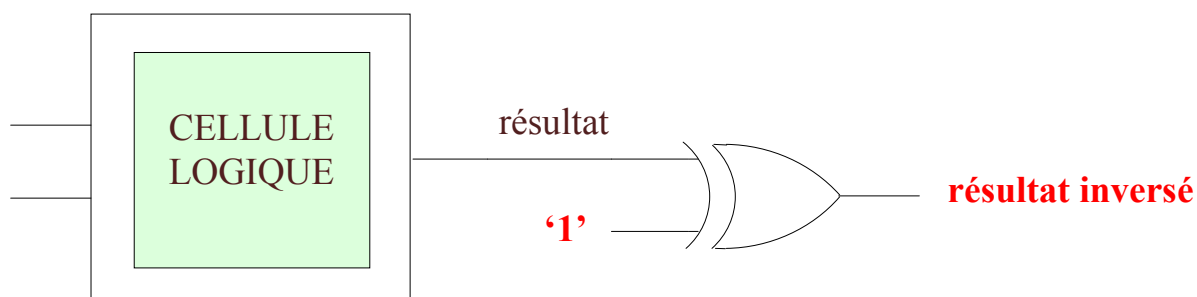
iii) émulation d'erreur

Sur le même principe que pour l'explication du principe de masquage d'erreur, nous allons utiliser la table de vérité de la porte XOR :

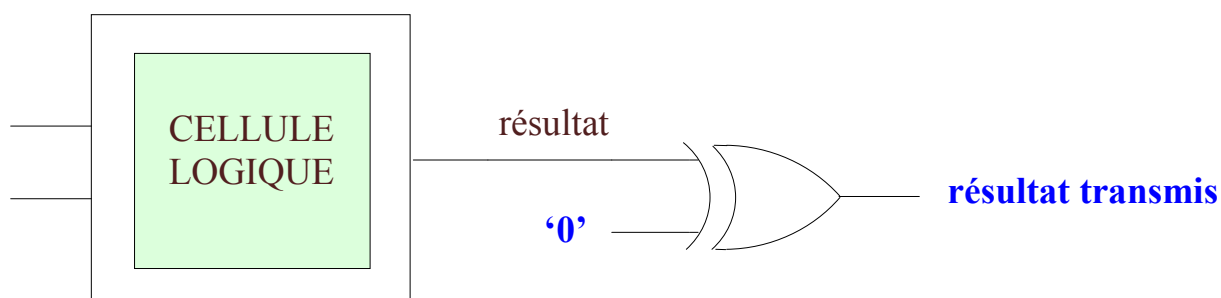
a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

} (b) est "transmis"
} (b) est INVERSÉ

Cette porte XOR est aussi appelée inverseur commandé car le fait de mettre une entrée (a) au niveau logique '1' permet de recopier l'inverse de l'autre entrée (b) en sortie :



...mais le niveau '0' étant ici aussi élément neutre de l'opération logique XOR, le fait de mettre une entrée (a) à '0' permet la "transmission intacte" de l'autre entrée (b) en sortie :



Maintenant que nous avons détaillé la technique utilisée pour l'émulation d'une injection d'erreur, nous pouvons développer le choix des nœuds particuliers évoqué précédemment.

iv) application de l'émulation d'erreur

Les applications de ce travail sont nombreuses, l'une d'elles est la conception de circuits *fiabiles*, depuis la description comportementale jusqu'au dessin de masque (layout).

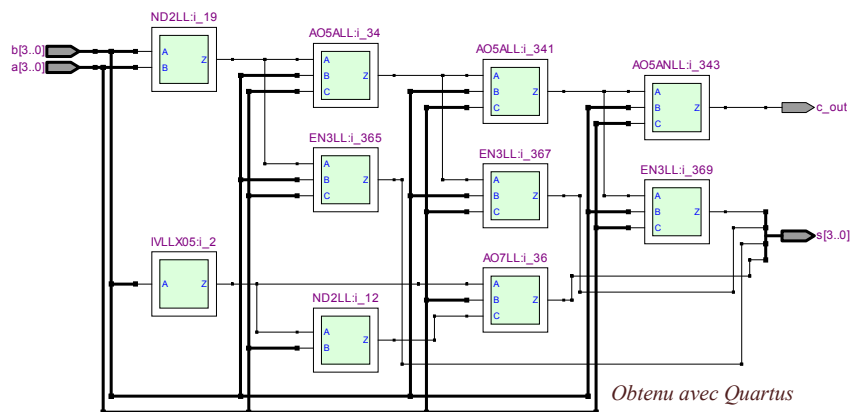
Pour ce faire, l'outil de synthèse utilisé, cadence, va générer une description structurée du circuit voulu. Cette description est composée d'instanciations de portes logiques plus ou moins complexes. Ces composants sont décrits et répertoriés dans les bibliothèques qui dépendent du fondeur choisi pour la conception finale du circuit intégré. Il peut s'agir de cellules logiques simples comme des inverseurs mais elles peuvent aussi être plus complexes et équivalentes à plusieurs portes logiques, souvent entre 2 et 5 (voir chapitre 3)d Résultats).

On considère donc ces cellules comme granularité minimale par soucis de réalisabilité de cette étude. En effet, on verra plus tard qu'on ne peut par exemple pas caractériser exhaustivement un opérateur 64bits en moins de plusieurs siècles avec une granularité plus fine.

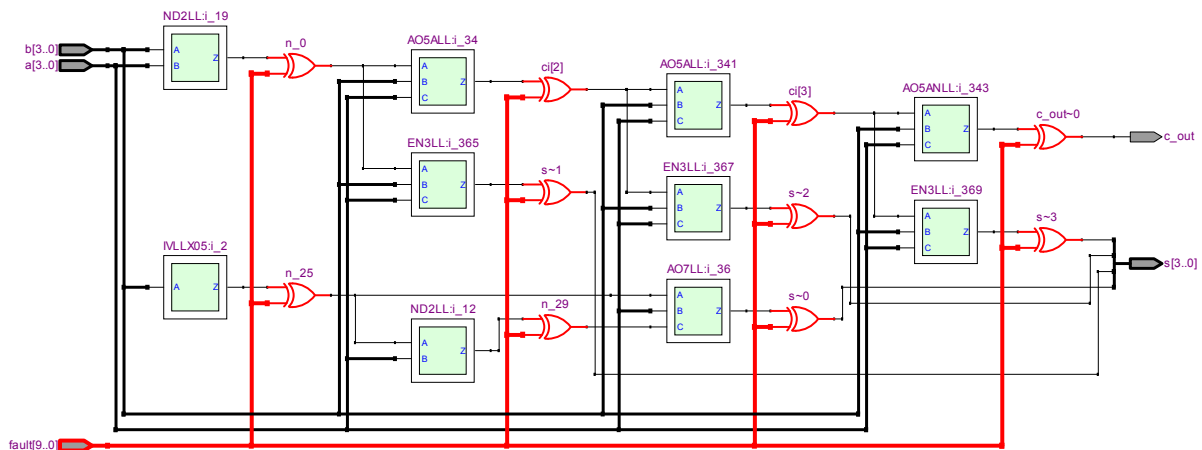
Quoi qu'il en soit, considérer qu'une erreur intervient en sortie d'une porte logique "classique" n'est déjà pas plus précis que de raisonner au niveau transistor non plus. Cette étude est donc limitée à ce niveau mais on sait que cette limite reste largement représentative car on peut correctement estimer la probabilité d'erreur de chacune de ces cellules logiques [10], [11].

Les nœuds choisis, évoqués précédemment pour nos injections de fautes, sont donc simplement la sortie de chacune de ces cellules logiques qui constituent nos opérateurs à tester.

Nous pouvons voir ci-dessous un exemple concret, l'additionneur à propagation de retenue 4bits :



Si on observe maintenant la représentation RTL (register transfer level) du composant de test ci-dessous, on constate qu'il est exactement le même mais on a incrusté les émulateurs d'injection d'erreurs à chaque sortie de cellule logique :



On peut maintenant injecter des erreurs aux nœuds désirés, la présence d'un '1' dans le **vecteur erreur (fault)** permettant donc une émulation de faute en un nœud voulu.

c) Expérimentations et réalisations

Maintenant que nous avons exposé le principe de l'évaluation de fiabilité nous pouvons développer les explications de l'aspect fonctionnel.

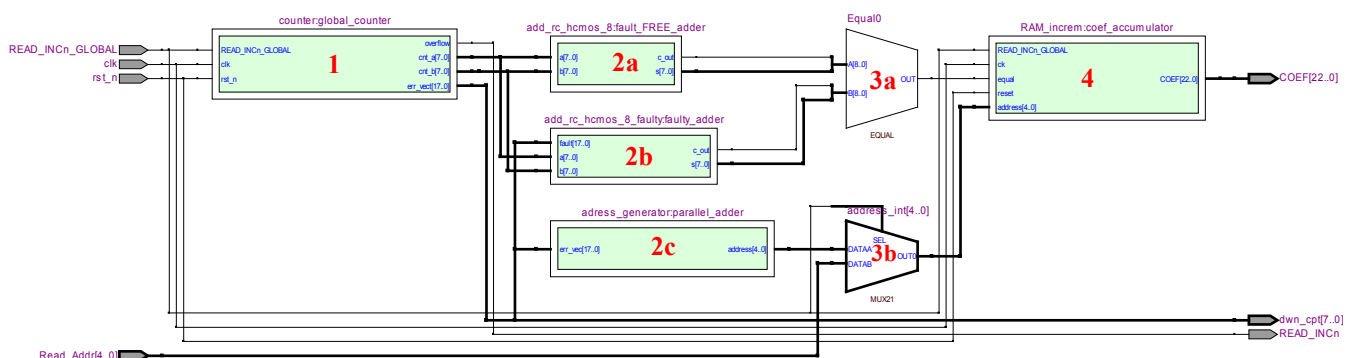
Pour s'imprégner de la problématique, nous avons commencé par concevoir une plateforme de caractérisation exhaustive à l'aide de description graphique au sein de Quartus (fichier .bdf).

Cette méthode de conception utilise des composants de librairie Altera qui étaient tout à fait convenables, mais leur assemblage présentait un lourd inconvénient. Effectivement, il aurait été très fastidieux de modifier tous les paramètres de ses composants lorsque l'on aurait eu à caractériser un autre opérateur que celui utilisé pendant cette première étape. Nous ne détaillerons donc pas cette description graphique mais elle est toutefois disponible en Annexe a.

En effet, après avoir effectué cette validation de fonctionnement du procédé, nous avons donc recommencé entièrement cette description en VHDL générique (on peut en trouver toutes les descriptions VHDL en Annexe a)

i) caractérisation exhaustive

Nous allons ici présenter le principe de fonctionnement du système dénombrement d'erreurs exhaustif en s'appuyant sur le schéma de synthèse suivant :



Lorsque le composant **1** reçoit l'ordre de démarrer la caractérisation, il génère :

- les mêmes entrées pour les composants **2a** et **2b** (opérateur idéal et opérateur sous test)
- un vecteur erreur pour le composant **2b** et **2c** (l'opérateur sous test et le calculateur d'adresse).

Le composant **2c** (calculateur d'adresse) compte le nombre de '1' du vecteur erreur. Ce nombre correspond à la quantité d'erreurs injectées dans l'opérateur sous test.

Le composant **3a** compare les résultats fournis par les opérateurs idéals et sous test. Si leur valeurs est identique, un signal de contrôle est délivré au composant **4**. Ce signal correspond à la détection d'un masquage d'erreur.

Dans notre choix de conception, la quantité d'erreurs correspond à l'adresse d'un compteur de masquages (composant **4**). L'incréméntation de ce compteur est donc autorisé par le signal de contrôle en sortie du composant **3a**.

En effet, le principe de cette caractérisation est de dénombrer le nombre de masquage selon le nombre d'erreurs injectées. Ces compteurs sont donc aussi nombreux que les bits du vecteur erreur, mais également aussi nombreux que les cellules logiques de l'opérateur testé.

Comme évoqué dans la présentation de ce modèle, le calcul de la fiabilité est assez complexe et prend en compte la probabilité d'avoir n erreurs. Il faut donc aussi connaître le nombre de masquages avec n erreurs (pour n allant de 0 à la taille du vecteur erreur).

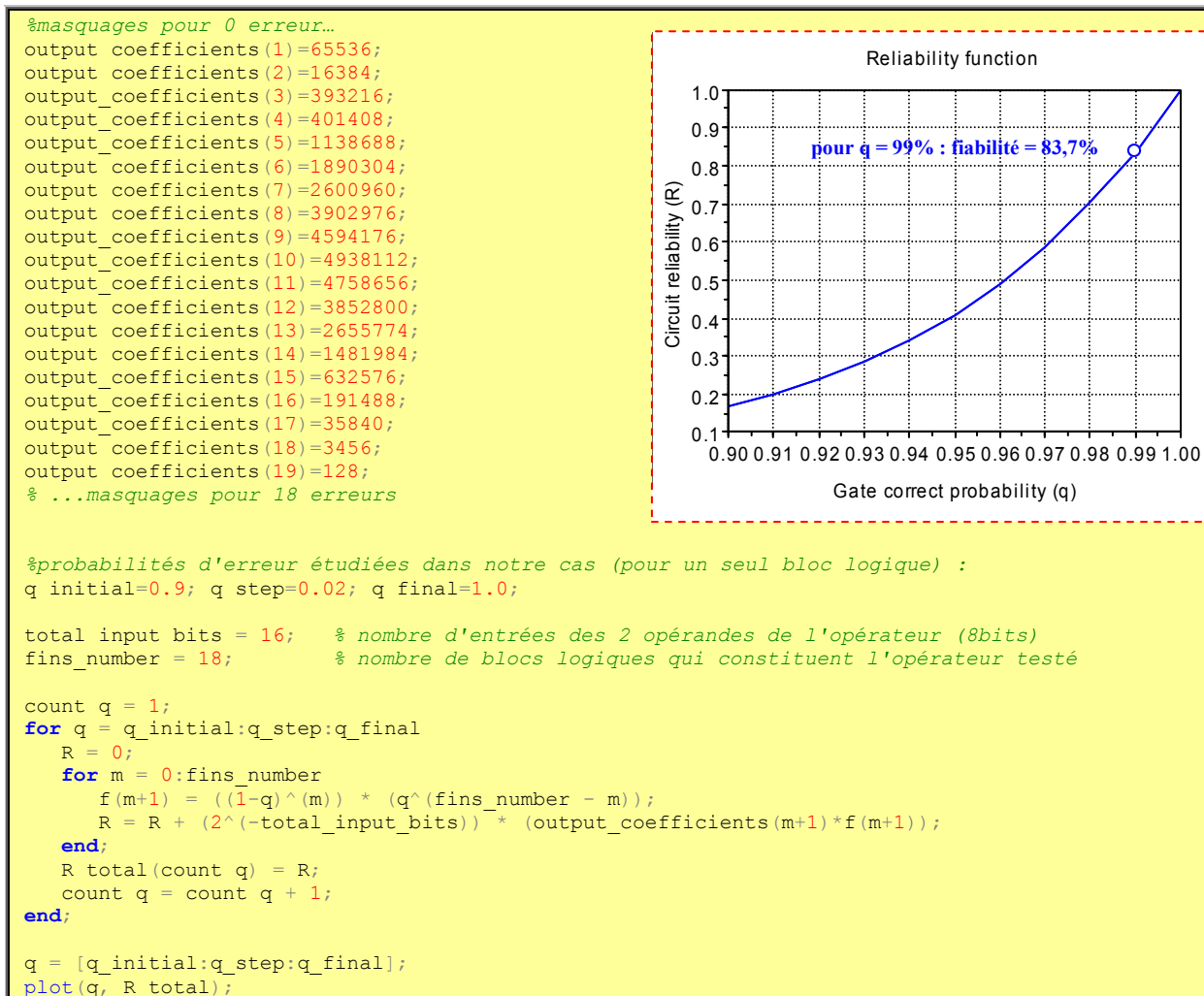
Lorsque le composant **1** a parcouru toutes les valeurs possibles pour les entrées et les erreurs, un signal est émis pour autoriser le rapatriement des valeurs mémorisées dans le composant **4**. Ce rapatriement est géré par le programme de contrôle du processeur embarqué dans le FPGA. Le processeur fournit donc les adresses des éléments mémorisés pour qu'ils lui soient envoyés.

Après cette étape d'incréméntation (ou écriture), le composant **3b** (un multiplexeur) permet donc de sélectionner l'adresse de lecture et la fournit au composant **4**.

Pour le rapatriement de ces données depuis le FPGA vers l'ordinateur, le processeur embarqué utilise donc le protocole UART par l'intermédiaire de l'interface JTAG.

Les valeurs récupérées sont donc inscrites dans un fichier et affichées sur une console de debug.

Voici les coefficients obtenus, ainsi que l'algorithme calculant la fiabilité et permettant son tracé en fonction de q , la fiabilité d'un seul bloc logique :



Bien qu'idéale en représentativité, cette méthode est souvent irréalisable car trop longue. En effet, si on désire caractériser un opérateur 32 bits, il n'est pas impossible qu'il soit composé de 2000 blocs logiques. Pour générer toutes les entrées et erreurs possibles il faut donc 2^{2064} cycles d'horloge ce qui prendrait plus de $4,91 \times 10^{2481}$ millénaires avec notre FPGA (à 50Mhz).

Pour accélérer ce procédé, nous allons voir ci-après les compromis que nous avons expérimentés.

ii) caractérisation aléatoire

On vient de voir que, le nombre de blocs logiques augmentant exponentiellement avec la taille des opérands d'un opérateur, une caractérisation exhaustive est parfois impossible. Nous allons donc présenter ici le principe de caractérisation aléatoire, qui permet d'accélérer la mesure du taux de masquage d'erreur.

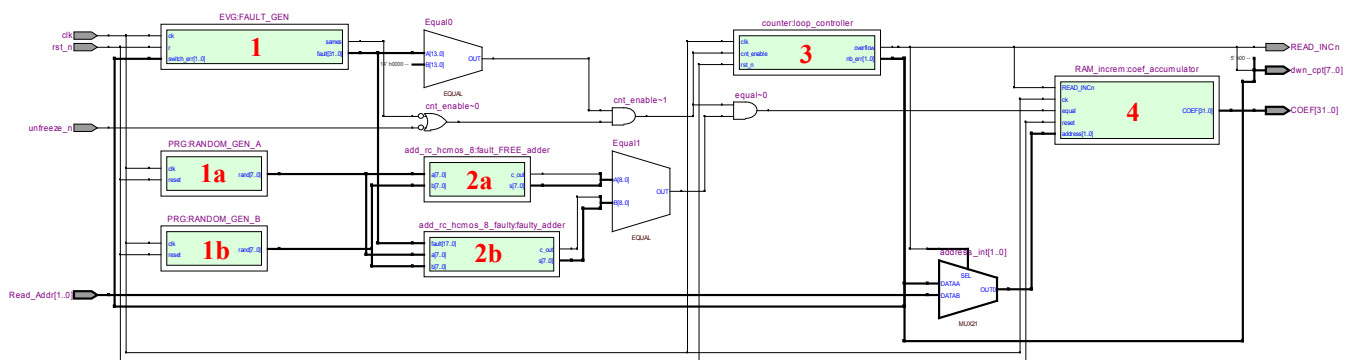
Il a été montré qu'une caractérisation avec un nombre réduit d'erreurs simultanées peut être assez représentatif [2]. Plus la fiabilité d'un seul bloc logique est grande, moins le nombre d'erreurs simultanées à tester est utile. Nous avons donc limité les injections d'erreurs simultanées au nombre de 4 et le nombre total de tests pour chaque cas a été proposé à 2^{20} (environ un million).

Conscient de la constitution d'un FPGA, nous avons utilisé le principe de décodage par Look Up Table pour générer nos vecteurs erreurs. L'idée était de créer aléatoirement une valeur binaire possédant entre un et quatre '1'. Nous avons donc utilisé des adresses générées aléatoirement pour mettre à '1' des bits du vecteur erreur (on rappelle que ce '1' dans le vecteur erreur permet une émulation d'erreur à un nœud donné du circuit).

Un signal de commande permet de sélectionner le nombre d'erreurs voulues, ce signal étant simplement constitué des 2 bits de poids fort d'un compteur 22bits.

Ce compteur permet donc d'équilibrer le nombre de tests voulus (ici 2^{20}) pour chaque nombre d'injections d'erreurs simultanées.

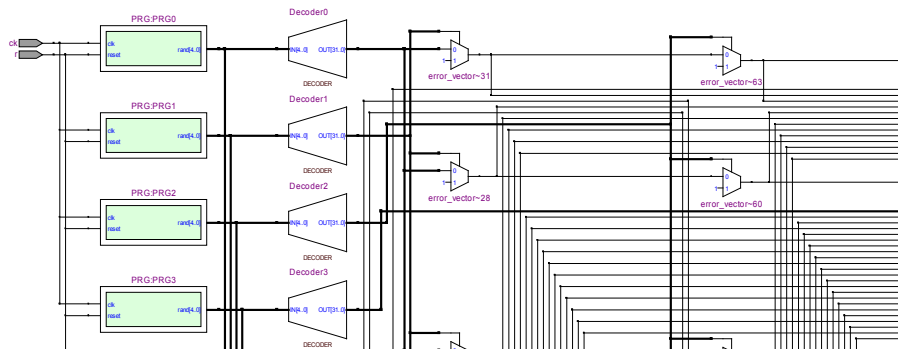
Voici un aperçu du schéma de synthèse de ce système :



Le composant **1** est le générateur de vecteur erreur pour l'opérateurs sous test (**2b**).

Les composants **1a** et **1b** génèrent des entrées aléatoires pour les opérateurs (**2a** et **2b**).

Le composant **3** (compteur) est donc aussi générateur d'adresses pour le composant de dénombrement de masquages (**4**).



Le composant **1** est donc composé de 4 générateurs aléatoires qui permettent l'injection d'1 à 4 erreurs comme expliqué précédemment.

Les résultats de ce système de caractérisation n'ont finalement pas été si satisfaisants dans le sens où les valeurs sélectionnées aléatoirement ne peuvent évidemment pas être autant représentatives que la méthode exhaustive avec des quantités de tests si réduites.

L'intérêt étant d'accélérer le processus de dénombrement des masquages, il aurait été ridicule de ne diviser que par 2 le nombre de valeurs testées. Pour une vraie représentativité ce n'aurait pourtant pas été trop.

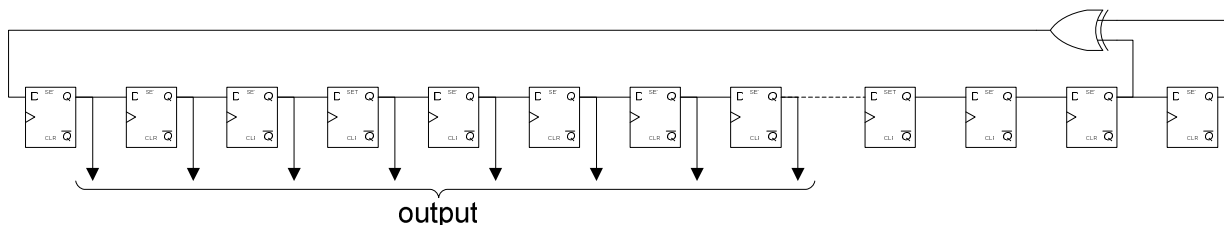
En effet, ce système présente deux imperfections :

- Si par exemple 2 adresses générées aléatoirement sont identiques, on ne peut avoir le nombre de '1' voulu dans le vecteur erreur.
- Si l'adresse générée aléatoirement est supérieure au nombre de bits du vecteur erreur, alors on peut se demander quel bit mettre à '1' sans déséquilibrer l'uniformité de la distribution des valeurs générées. Par exemple, dans le cas de l'additionneur 8bits à propagation de retenue, le vecteur erreur fait 18bits de large. Pour générer une des valeurs entre 0 et 17 il faut 5 bits mais cela donne presque une fois sur deux des valeurs supérieures à 17.

Pour compenser ces imperfections il faut répéter la génération aléatoire et donc attendre un coup d'horloge à chaque fois. Tous ces cas annulés sont parfois trop nombreux et bloquent littéralement le processus de test.

À ces deux problèmes s'ajoute celui de la linéarité de la distribution des valeurs générées. Pourtant, le générateur aléatoire utilisé, un *Linear feedback shift register* est assez classique et la linéarité de sa distribution n'est pas mauvaise [3].

Voici le schéma de notre *pseudo random number generator* sur m bits : (exemple avec $m = 8$)



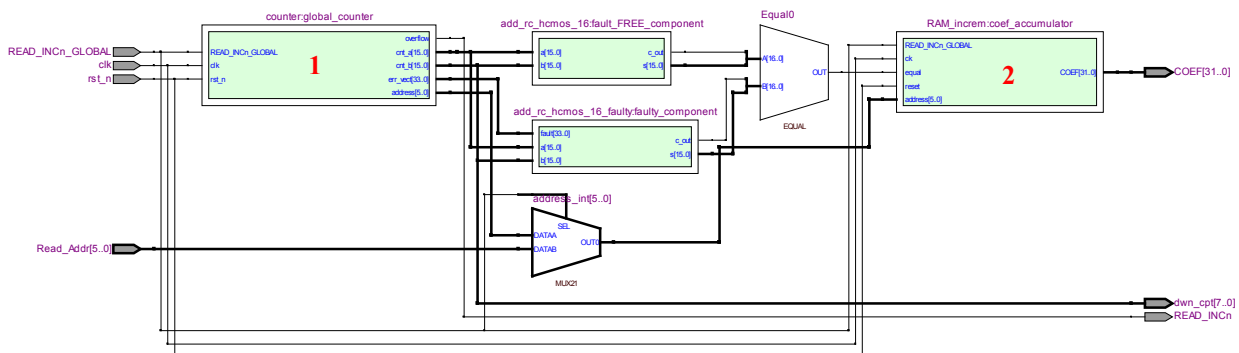
Pour augmenter l'aspect aléatoire des valeurs générées, on n'utilise en sortie que les m bits de poids faible (on ajoute 4 bascules pour obtenir 16 fois plus de combinaisons aléatoires).

Malgré tout, les calculs effectués par cette méthode n'étant pas concluants, nous avons développé une autre plateforme de caractérisation. Il est tout de même envisagé de continuer à chercher les améliorations possibles mais le temps disponible est limité.

iii) caractérisation à une erreur

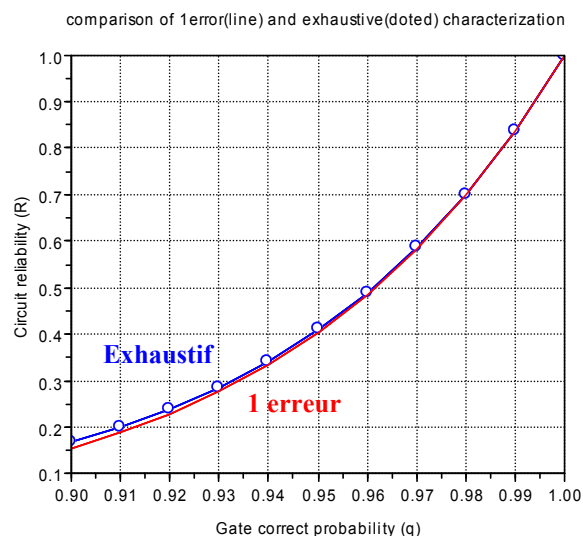
Il a été montré que si l'on mesure le taux de masquage avec toute entrée possible pour toute injection d'une erreur possible, alors on peut obtenir une caractérisation représentative [2]. Il faut tout de même prendre en compte un paramètre qui peut paraître évident, la fiabilité d'un bloc logique ne doit pas être trop faible (au moins 90%). Les conditions de fonctionnement de la quasi-intégralité des circuits électroniques permettent de constater une fiabilité telle que bien moins d'une erreur sur plusieurs milliers de cas intervient. Une fiabilité de 90% correspondant à un taux d'erreur de 10%, la marge de fonctionnement est large.

Le fonctionnement étant presque le même que pour les précédents systèmes, nous n'allons que le présenter succinctement:



Le composant **1** génère les entrées de façon exhaustive, ainsi que les erreurs. Le numéro du bit d'erreur est aussi fourni en sortie, il sert d'adresse au composant **2**. En effet, il est intéressant de pouvoir savoir quelles injections d'erreurs ont provoqué le moins de masquages, on peut ainsi repérer les parties les plus sensibles du circuit et donc les renforcer de façon ciblée.

On rappelle donc que si l'on ne considère que les cas réels, les fiabilités pour une porte logique sont généralement de bien plus de 99% (donc bien moins de 1% d'erreur). Les tests que l'on a effectués confirment que la différence entre caractérisation exhaustive et caractérisation à une erreur est très mince, voici un exemple avec un additionneur 8 bits :

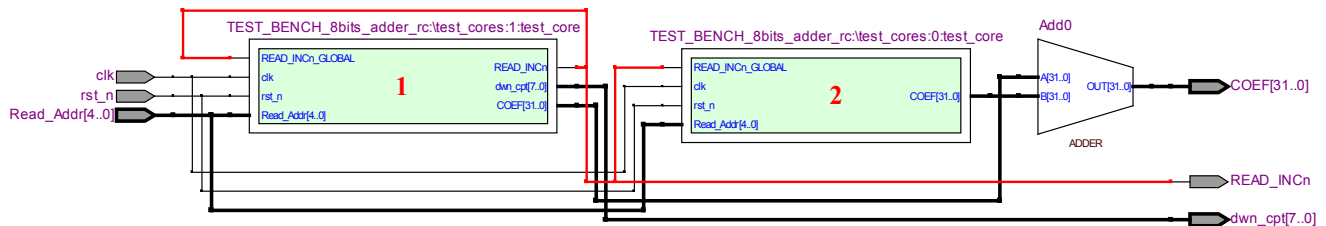


On constate donc que dans les zones réalistes, ce procédé est tout à fait représentatif. Ayant aussi l'avantage de révéler les nœuds sensibles, cette méthode est bien plus qu'un bon compromis.

iv) caractérisation "multi cœurs"

Alors que le temps est au bi processeur, quadri processeur, voire plus, il aurait été dommage de ne pas profiter de la surface offerte par le FPGA pour accélérer les mesures. Nous avons donc voulu rentabiliser la possibilité d'effectuer des calculs parallèles en démultipliant les tests benches. Ce procédé permettant une répartition des tâches intelligente, on a donc décrit un système permettant de démultiplier "à volonté" les plateformes de caractérisation. On verra qu'une limitation de surface est bien sûr rapidement inévitable, aucun FPGA n'est sans fin.

Voici le schéma de synthèse du système à deux cœurs:



Les composants **1** et **2** effectuent les mêmes tâches que les test benches précédents, ils se sont juste réparti les valeurs à traiter en deux ensembles de même taille pour finir deux fois plus vite.

Le composant **1** s'occupant de la 2^{ème} moitié des valeurs à traiter, l'overflow de son compteur interviendra au moment où toutes les valeurs auront été parcourues, c'est donc lui qui sert de cœur maître (il émet les signaux de contrôle et permet l'aperçu du temps de calcul restant).

A la fin des calculs effectués, on additionne simplement les coefficients accumulés adresse par adresse et on les restitue en sortie.

Au lieu de 6 minutes pour une caractérisation exhaustive de l'additionneur 8bits évoqué, on n'aura besoin que de la moitié. Évidemment, cette différence de 3 minutes ne se fait pas tant sentir mais pour un opérateur de taille plus importante, ce facteur 2 peut facilement représenter plusieurs semaines de différence.

On a donc conçu un procédé permettant la démultiplication selon un paramètre générique. Les derniers tests effectués on permis de synthétiser 32 cœurs, on effectue les calculs complets en quelques secondes dans ce cas-là. La surface utilisée devenant de moins en moins négligeable, le temps de compilation, lui aussi, devient une limite. Une comparaison de ces différentes architectures et leurs limites est disponible en Annexe b.

Note : Le calcul de fiabilité selon le modèle PBR ne sera pas expliqué plus profondément dans ce mémoire, mais il est aisément possible de le trouver dans la littérature [2].

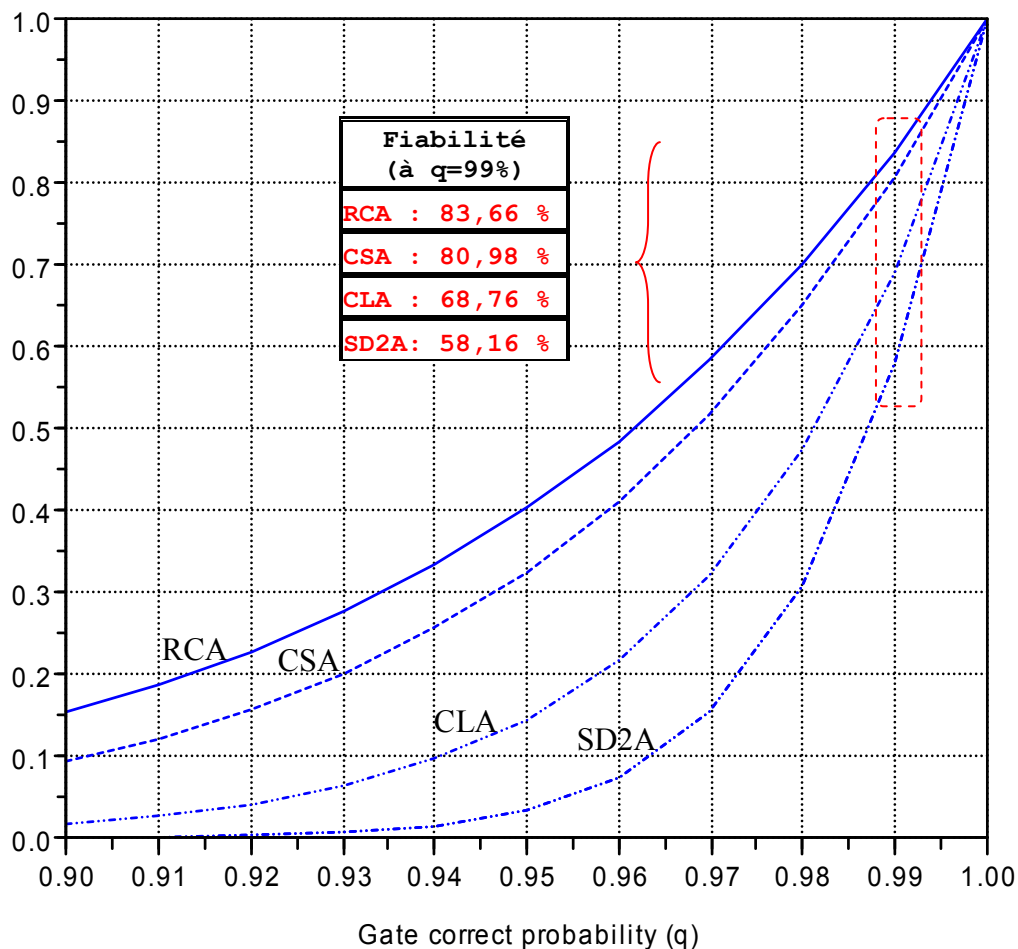
d) Résultats

Grâce à cette plateforme d'évaluation de fiabilité améliorée, un assez grand nombre d'opérateurs a été caractérisé. On ne présentera ici que les résultats de certains des plus courants pour des tailles d'opérandes de 4 bits, 8 bits et 16bits :

- Les trois additionneurs classiques caractérisés sont basés sur la propagation de retenue (Ripple Carry), la sélection de retenue (Carry Select) et l'anticipation de retenue (Carry Lookahead).
- Un additionneur à temps constant a aussi été caractérisé, basé sur le système de représentation redondante des données du type nombres signés (Signed Digits).
- Enfin, un multiplieur a aussi été caractérisé ; son architecture est de type Booth.

Comparaison de la fiabilité des 4 additionneurs évoqués sud 4 bits : (caractérisation à 1 erreur)

Comparison of RCA(line) CSA(dashed) CLA(dotted) SD2A(dashed-dotted)



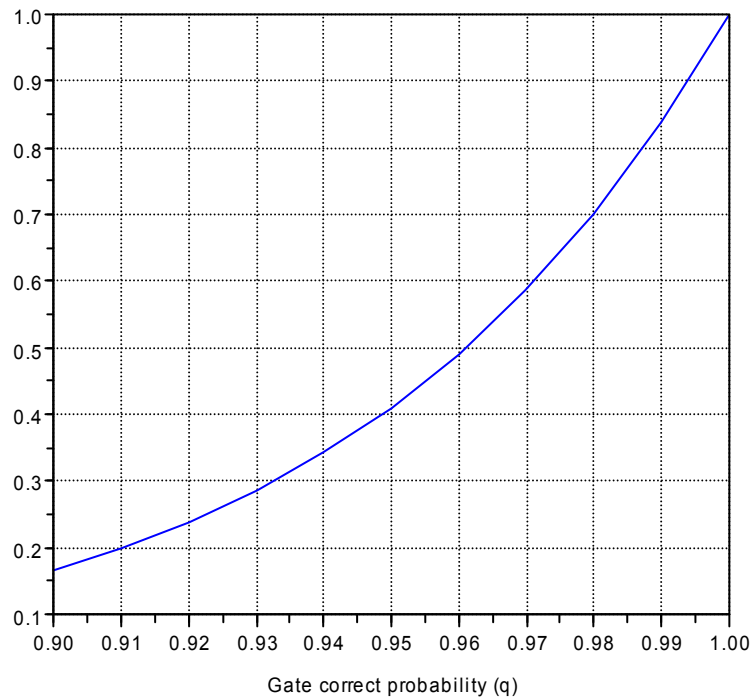
On compare les fiabilités à 99% car c'est le minimum réaliste (ça correspondrait à une probabilité d'erreur dans le circuit d'un sur cent). En général, on a donc "beaucoup" plus que 99%.

Le « Ripple Carry Adder » est donc bien plus fiable que le « Signed Digits 2 adder », on verra d'ailleurs dans les détails suivants que le nombre de cellules logiques constituant un opérateur est un paramètre à surveiller.

Pour illustrer la similarité du comportement de la fiabilité selon l'opérateur, nous allons détailler la caractérisation des additionneurs comparés précédemment, mais sur 8 bits :

Résultats: RCA 8bits
(Caractérisation exhaustive)

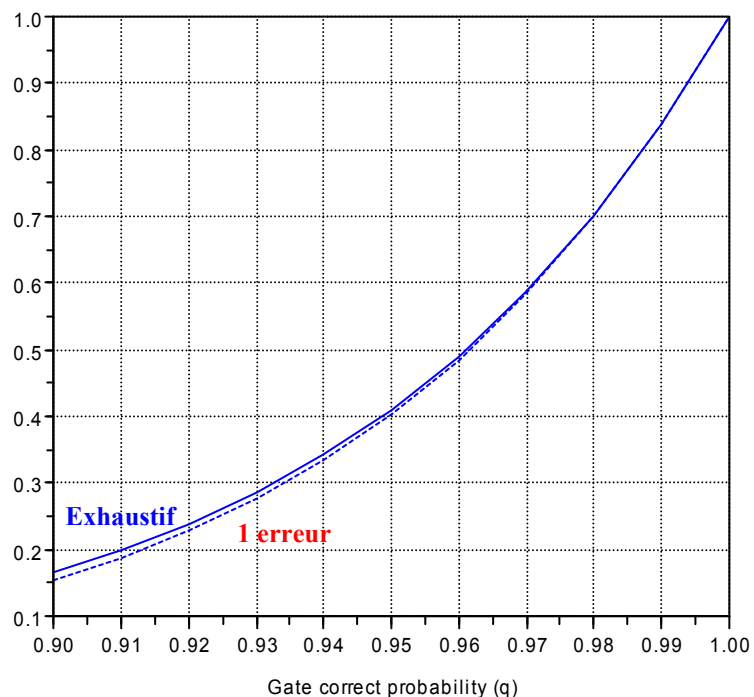
```
R_total =
0.1671273
0.1998464
0.2390528
0.2860095
0.3422169
0.4094554
0.4898345
0.5858513
0.7004574
0.8371374
1.
```



Comparaison de la caractérisation avec injection d'une seule erreur (pointillés):

Résultats: CSA 8bits
(Caractérisation à une erreur)

```
R_total =
0.1542639
0.1876517
0.2277828
0.2759239
0.3335622
0.4024408
0.4845992
0.5824200
0.6986819
0.8366211
1.
```

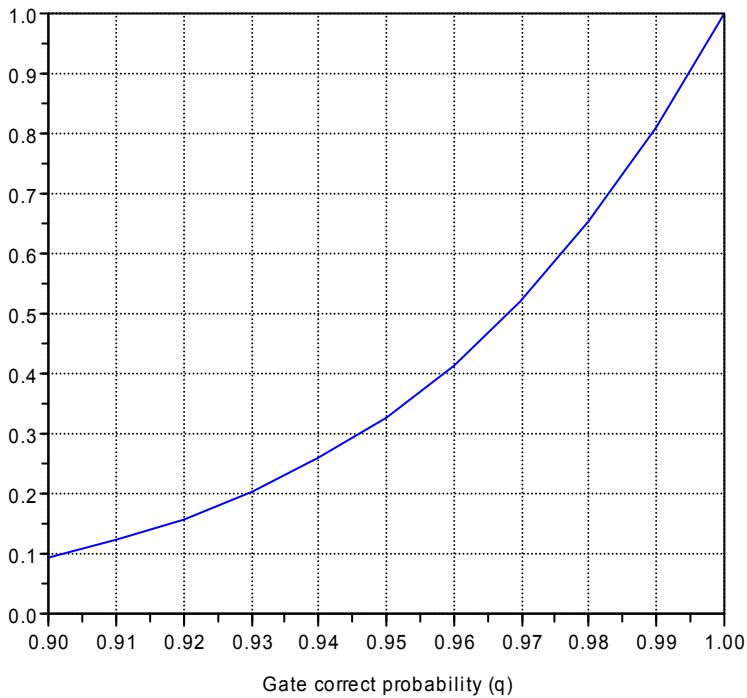


Comme évoqué, on constate une différence infime lorsque les probabilités d'erreur sur une porte sont élevées (= fiabilité d'une porte faible).

Par contre, pour des probabilités d'erreur sur une porte plus réalistes ($q=99\% \Rightarrow 1\%$ d'erreur) on a des fiabilités de **83,66%** en exhaustif et **83,71%** avec 1 erreur ce qui représente une approximation tout à fait satisfaisante.

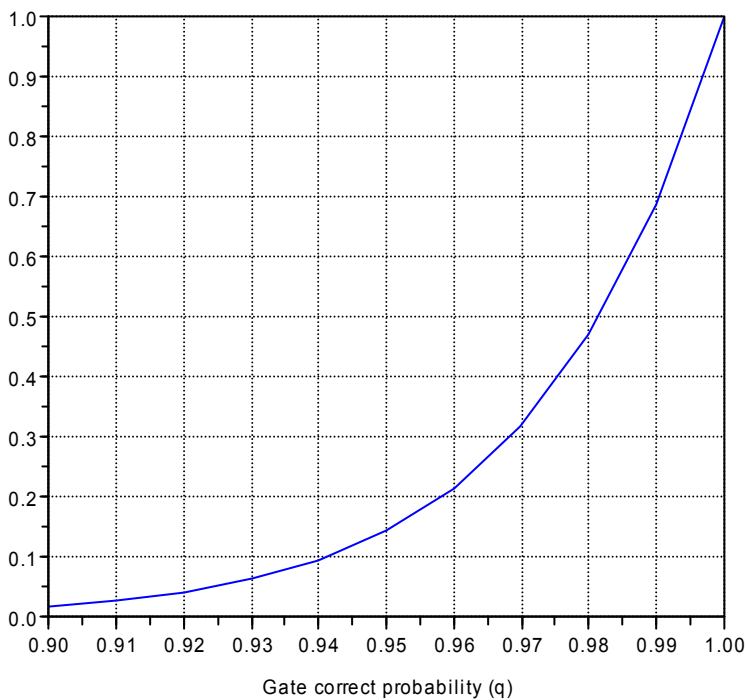
Résultats: CSA 8bits

```
R_total =
0.0943457
0.1222650
0.1576791
0.2023532
0.2583886
0.3282588
0.4148397
0.5214249
0.6517192
0.8097956
1.
```



Résultats: CLA 8bits

```
R_total =
0.0171710
0.0265865
0.0408905
0.0624702
0.0947982
0.1428830
0.2138852
0.3179467
0.4692877
0.6876304
1.
```

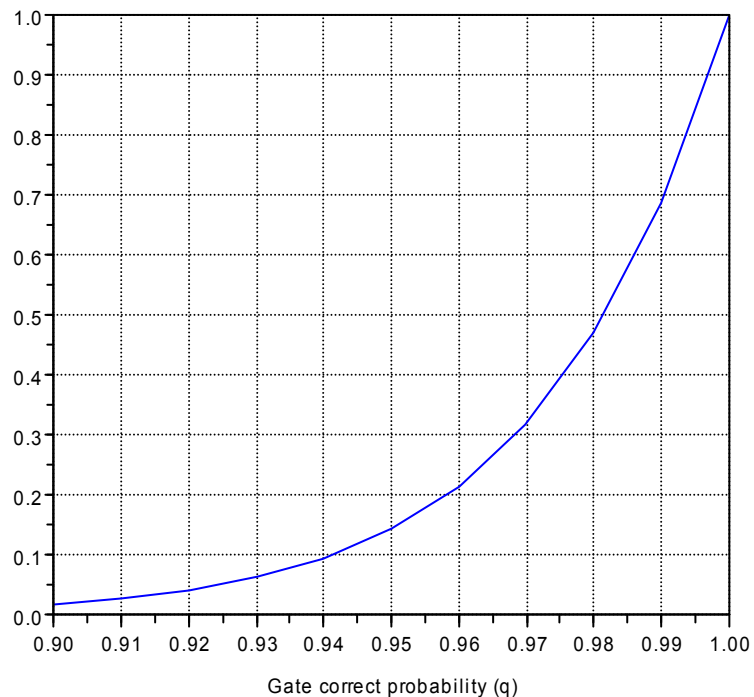


Résultats: SD2 8bits

```

R_total =
0.0171710
0.0265865
0.0408905
0.0624702
0.0947982
0.1428830
0.2138852
0.3179467
0.4692877
0.6876304
1.

```



Si l'on se focalise maintenant sur quelques détails de la caractérisation pour l'additionneur RCA, on peut étudier le dénombrement des masquages selon le rang de l'erreur injectée :

- modèle 16bits : (temps de calcul = 730.31s)

```

ERROR_RANK (1)=0;
ERROR_RANK (2)=1073741824;
ERROR_RANK (3)=0;
ERROR_RANK (...)=0;
ERROR_RANK (34)=0;

```

- modèle 8 bits: (temps de calcul négligeable)

```

ERROR_RANK (1)=0;
ERROR_RANK (2)= 16384;
ERROR_RANK (3)=0;
ERROR_RANK (...)=0;
ERROR_RANK (18)=0;

```

- modèle 4 bits: (temps de calcul négligeable)

```

ERROR_RANK (1)=0;
ERROR_RANK (2)=0;
ERROR_RANK (3)=64;
ERROR_RANK (4)=0;
ERROR_RANK (...)=0;
ERROR_RANK (10)=0;

```

On constate que tous les masquages interviennent au même rang d'injection d'erreur, et ce quelle que soit la taille testée de cet opérateur. En observant sa description, on peut retrouver le nœud qui provoque ces masquages.

Pour des raisons de simplicité de lecture on étudiera la version 4 bits de ce RCA.

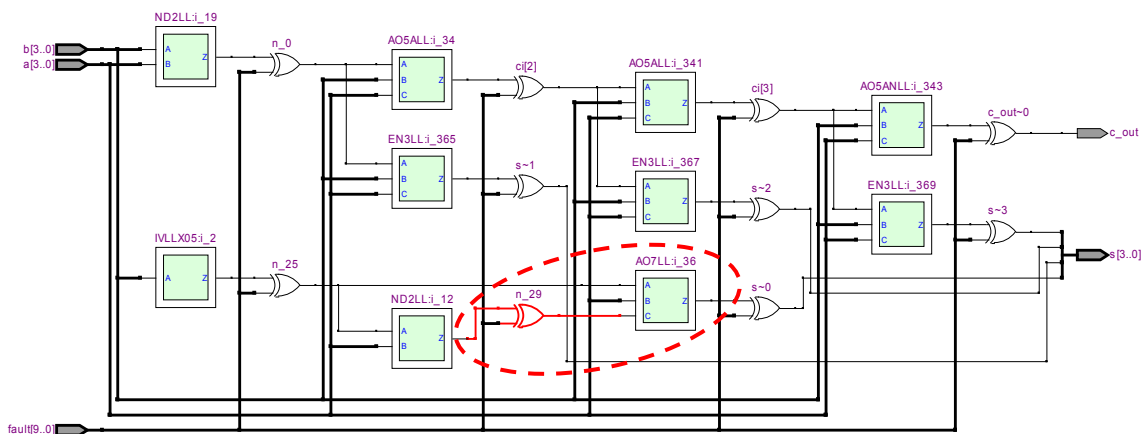
Opérateur RCA 4bits avec injection d'erreur, description VHDL :

```

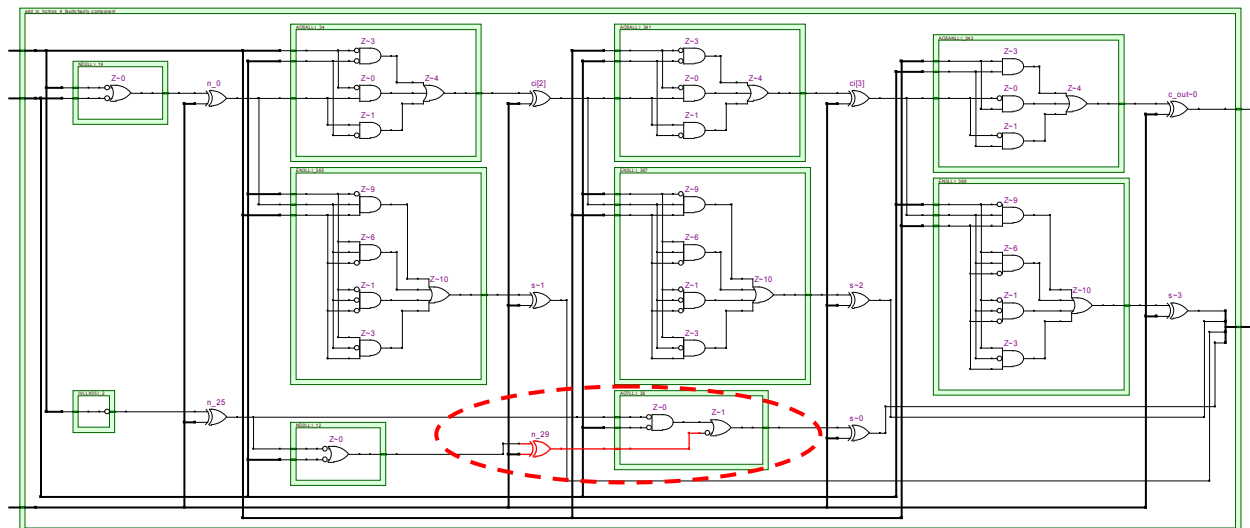
-- ...
i_36 : AO7LL
port map (A => n_25, B => a(0), C => n_29, Z => faulty_node(1));
-- ...
n_25 <= faulty_node(0) XOR fault(0);
s(0) <= faulty_node(1) XOR fault(1);
n_29 <= faulty_node(2) XOR fault(2);
s(1) <= faulty_node(3) XOR fault(3);
-- ...

```

Opérateur RCA 4bits avec injection d'erreur, description RTL :



Opérateur RCA 4bits avec injection d'erreur, description RTL détaillée :

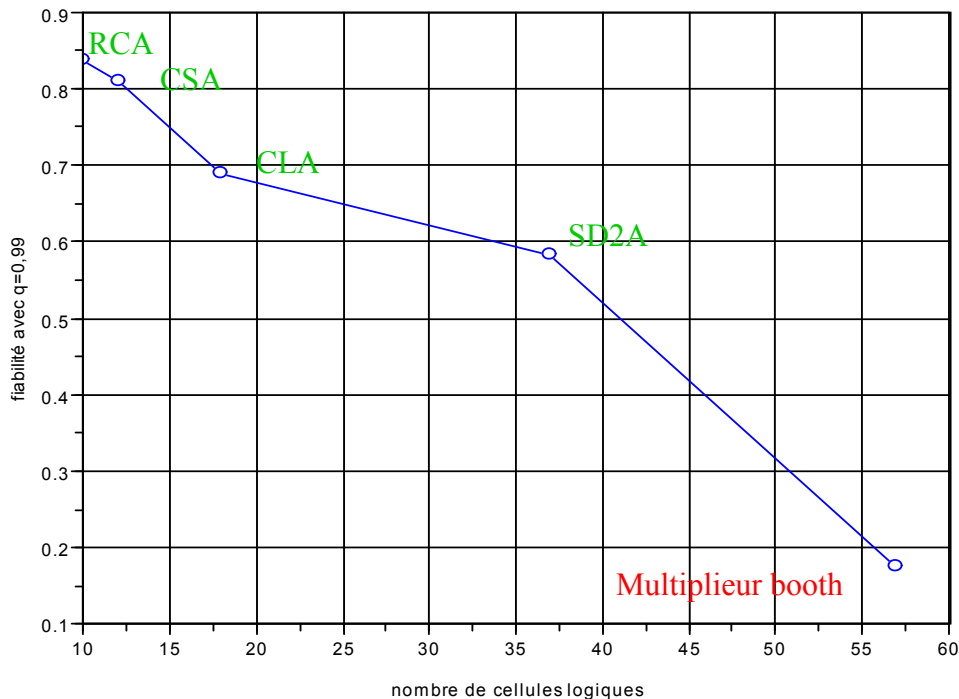


On injecte l'erreur en sortie du ND2LL:i_6 grâce au XOR n_29 commandé par le fil fault(2) et elle se répercute en entrée dans le bloc intitulé AO7LL:i_132.

Ceci laisse conclure que le composant "OU logique" intitulé Z~1 dans le bloc intitulé AO7LL:i_132 est celui qui permet les masquages dans ce circuit.

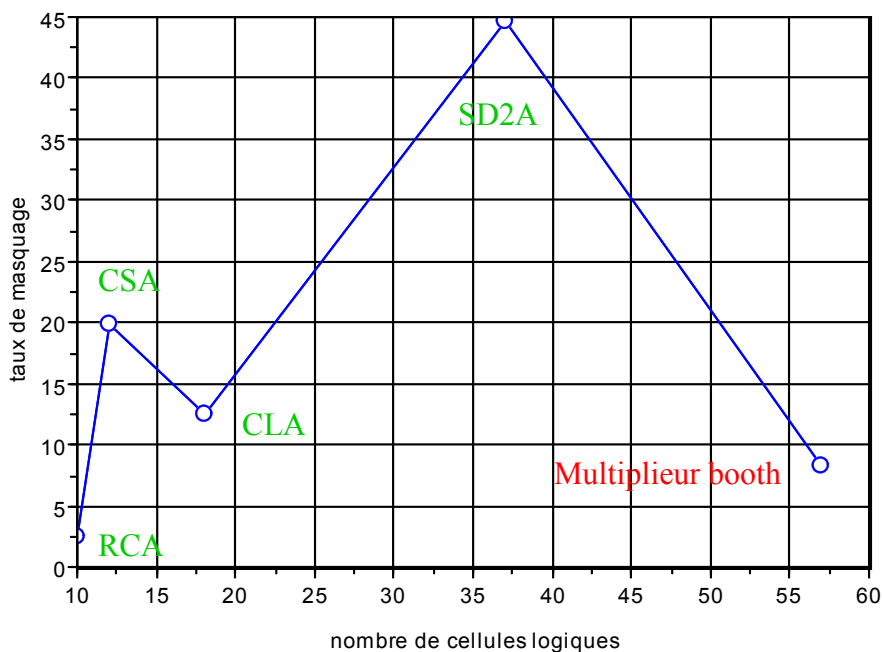
Ce type d'analyse peut permettre de révéler les parties robustes mais aussi de cibler les renforcements des parties sensibles. On verra en conclusion qu'il existe plusieurs façons d'effectuer ces renforcements.

Comparaison des 4 additionneurs et un multiplieur :



La comparaison avec ce multiplieur est un cas extrême pour appuyer la constatation suivante : Plus un opérateur est constitué d'un grand nombre de cellules logiques, plus sa fiabilité est réduite.

Pour correctement appréhender cette observation, il est important de bien faire la distinction entre fiabilité et taux de masquage. Le taux de masquage n'étant qu'un paramètre permettant d'obtenir la fiabilité, on peut illustrer cette différence en comparant le tracé précédent et celui qui suit :



On voit bien que malgré l'allure croissante du taux de masquage en fonction du nombre de cellules logiques des additionneurs, leur fiabilité décroît. Cette différence s'explique par la prise en compte de la probabilité d'erreur d'une seule cellule logique et le nombre de ces cellules logiques. Ce nombre est finalement clairement déterminant sur la fiabilité d'un composant numérique.

4) Conclusion

a) Difficultés

Les rares difficultés rencontrées pendant ce stage ont été rapidement arrangées et ne présentent majoritairement pas de grand intérêt. Je voudrais tout de même en dédier une au Professeur Patrick Garda, qui lors de ses précieux enseignements nous a tous impressionnés (et amusés) en nous prévenant avec un cas *presque* similaire :

- Lors de la conception du premier prototype de plateforme de caractérisation, tous les composants n'ont pas été simulés pour gagner du temps. Il a été évoqué que cette simulation aurait pris 14h mais elle aurait été grandement bénéfique.

En effet, à cause d'un overflow de la mémoire de dénombrement d'erreurs masquées, presque tous les coefficients rapatriés étaient faux. Cet overflow était dû à une mauvaise estimation du plus grand nombre de masquages possibles, la taille choisie était sur 18bits au lieu de 22.

Coût : 1 week-end de cauchemars.

- En 1996, le premier vol d'Ariane 5 est tenté pour placer en orbite deux satellites. L'un des paramètres de contrôle/calculs relatifs au moteur dépassa la valeur du plus grand short et ne peut être down-casté.

Il y a overflow en cascade et "plantage" généralisé, la fusée se désintègre après 40s de vol.

Coût : 500Million d'euros.



Décollage, puis explosion, d'Ariane 5 - © CNN

La morale de cette mauvaise expérience est qu'il vaut mieux ne pas être trop économe pour une conception de prototype. Les optimisations sont donc plus adéquates après vérification du bon fonctionnement du composant conçu (ce qui paraît finalement assez logique).

b) Perspectives

Une des principales observations permises par ce travail est que plus un opérateur est complexe, moins il est robuste. On a en effet constaté que le nombre de ses cellules logiques est un lourd poid sur sa fiabilité.

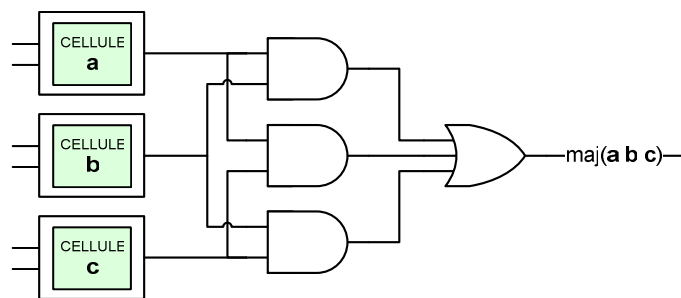
Ce constat permet une application directe, lors de la conception de système sensible aux erreurs, qui consiste simplement en la sélection d'opérateurs en fonction de leur fiabilité. Cette sélection permet une augmentation de la robustesse aux erreurs quelles que ce soient leurs causes possibles :

- fautes de fabrication (ou *stuck-at-faults*) ;
- radiations (rayons alpha, ions lourds, etc.) ;
- attaques volontaires (pour obtention d'informations sur le fonctionnement du système).

Il est aussi possible de renforcer de façon ciblée des zones sensibles de circuits particulièrement peu robustes, au niveau logique ou mieux, au niveau microélectronique.

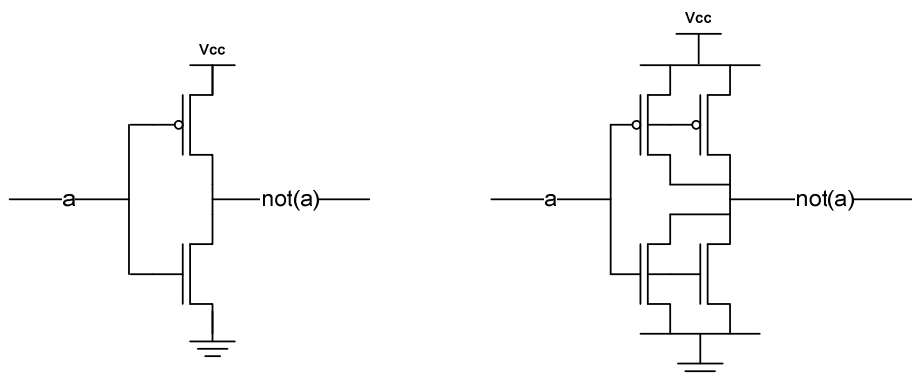
- D'un point de vue logique, plusieurs solutions sont proposées dans la littérature, l'une d'entre elles est la correction par majorité ou TMR [5], [7] (pour Redondance Modulaire Triple). En effet, il est possible d'effectuer une opération logique de façon différente, par exemple, trois fois. On peut sélectionner la majorité des résultats (binaires) obtenus grâce à un vote, en considérant qu'il est moins probable que deux erreurs interviennent plutôt qu'une.

Voici un schéma de fonctionnement du circuit de vote :



Cette solution présente un défaut important, la surface utilisée est au moins triplée. Il est aussi possible de détecter une erreur en comparant la parité des opérandes fournies et du résultat obtenu, mais cette solution augmente le temps moyen de calcul (en cas d'erreur, il faut répéter le calcul).

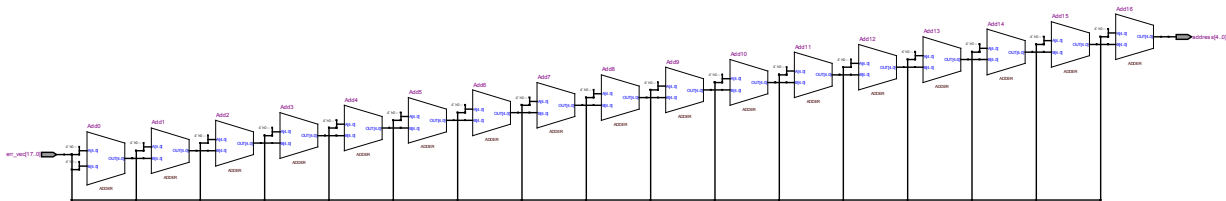
- D'un point de vue microélectronique, il a été observé que les blocs logiques proches des sorties sont plus sensibles (la probabilité que leur erreur soit masquée est moindre). Il est donc possible de renforcer de façon encore plus ciblée les zones importantes. Ce renforcement peut donc être effectué en démultipliant les transistors comme dans le schéma suivant :



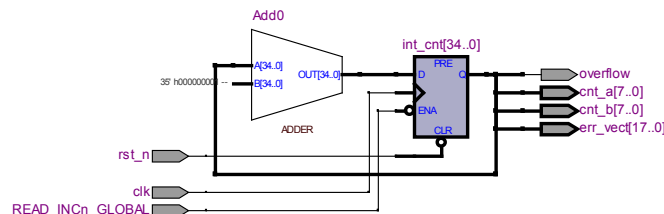
Il est aussi possible d'augmenter la tension d'alimentation de ces transistors, ou encore d'augmenter leur surface. Ces choix doivent encore être effectués selon des compromis en fonction des consommation et vitesse d'utilisation.

En se plaçant à moins long terme, le travail effectué pendant ce stage peut encore être optimisé. Il est par exemple possible de se rapprocher de mécanismes moins coûteux en surface pour le traitement de l'information du masquage. Quelques améliorations sont réalisables avant la fin de ce stage, en voici quelques exemples :

- Adress generator : peut être remplacé par un additionneur en arbre au lieu d'un additionneur cascadi (le temps de propagation est proportionnel au nombre de ses entrées).



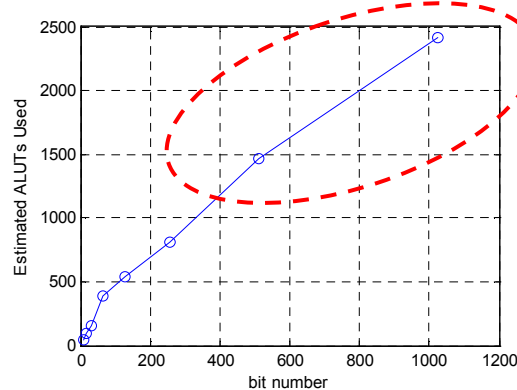
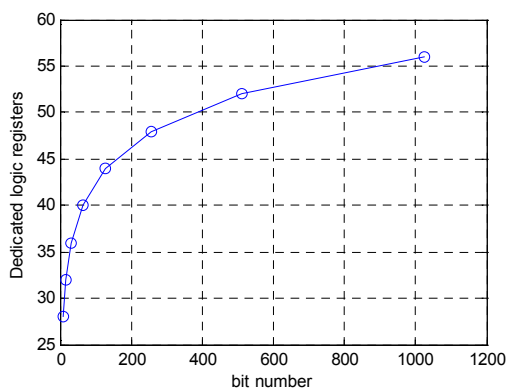
- Compteur : peut être remplacé par une cascade de diviseur de fréquences synchrones (l'additionneur implémenté est finalement inutile).



- Le schéma RTL du composant "RAM_increment" n'est pas représenté ici mais, étant composé du même type de compteurs que précédemment, on peut tout autant l'optimiser.

- Le problème de surface utilisée peut être posé pour le composant EVG (error vector generator). Ce générateur de 1 à 4 erreurs aléatoires sur N bits utilise des ressources matérielles de cet ordre :

Output error vector size N:	8 bits	16 bits	32 bits	64 bits	128 bits	256 bits	512 bits	1024 bits
Estimated ALUTs Used:	49	100	158	392	544	807	1465	2414
Dedicated logic registers:	28	32	36	40	44	48	52	56



- Enfin, dans la démultiplication des tests benches, il est aussi possible de réduire l'espace utilisé en évitant de démultiplier des éléments redondants. En effet, on pourrait n'instancier que le nombre de compteurs indispensables, et en faire autant pour l'additionneur référence (un seul peut suffire).

c) Épilogue

Si l'on résume le travail effectué lors de ce stage, on peut souligner plusieurs points importants:

- Accélération de mesures grâce au FPGA ;
- Systématisation grâce au VHDL générique ;
- Application concrète du codesign.

Ces points constituent une mise en application parfaite de l'ensemble des connaissances appréhendées lors de cette première année de master Sciences De l'Ingénieur mention Architecture et Conception de Systèmes Intégrés.

D'un point de vue personnel, il a aussi été intéressant d'avoir été mis à contribution dans un projet de recherche réel : la mise en oeuvre d'un outil d'évaluation de la robustesse d'opérateurs numériques. Cette étude a permis une participation à l'étude des compromis robustesse-coût-vitesse d'opérateurs numériques, qui pose un réel problème pour un avenir plus que proche : les technologies de conception d'ordre déca nanométrique.

Il a d'ailleurs été particulièrement enrichissant de pouvoir assister au Colloque System On Chip System In Package, où de nombreux projets de recherche passionnants ont été exposés. D'autres séminaires moins formels ont aussi été très instructifs comme ceux d'étudiants présentant leur travail annuel de thèse ou de fin d'études. Enfin, l'accès aux salles de travaux pratiques et laboratoires voisins m'a permis de découvrir d'autres domaines comme la cryptologie ou la recherche sur la conversion delta sigma (avec cour particulier et démonstration sur FPAA).

Au jour où ce mémoire est écrit, ce stage n'est pas encore fini. Les quelques étapes à accomplir avant cette fin résident entre autre en:

- La création d'un WIKI pour transmettre les connaissances plus techniques mises en place pour l'équipe de relais (qui continuera ce travail) ;
- L'optimisation de tout ce qui a été décrit de façon uniquement fonctionnelle comme ce qui a été évoqué en partie b) perspectives. Il est aussi possible, par exemple, de pipeliner certaines parties combinatoires pour utiliser les PLL du FPGA et ainsi accélérer encore le processus de caractérisation.
- La caractérisation de composants qui n'ont pas encore été créés ;

Tout ce travail fut un réel plaisir, la possibilité de contribuer à l'enrichissement d'une connaissance collective, de participer à un travail dans une équipe motivée et motivante, de découvrir ce monde fascinant et multiculturel qu'est celui de la recherche française...

Je voudrais donc sincèrement remercier Julien Denoulet qui m'a parfaitement armé pour combattre la théorie des opérateurs numériques que j'ai utilisée dans ce stage ; mais aussi pour ses patientes recommandations d'orientation de mes études, qui s'approchent de plus en plus du doctorat.

Pour ces mêmes raisons, je voudrais aussi exprimer ma profonde reconnaissance envers Bertrand Granado sans qui je n'aurais simplement jamais découvert la passion que j'éprouve aujourd'hui pour l'électronique numérique.

Je veux enfin remercier chaleureusement Lírída Naviner pour la qualité de son encadrement, scientifique mais aussi humain. Pour tous ses précieux conseils et pour la confiance qu'elle m'a accordée en m'accueillant dans son équipe, merci.

5) Annexe

a) Programmes

L'équipe de travail sur ce domaine étant multiculturelle, les descriptions qui suivent sont commentées en anglais pour un souci de compréhension par tous les membres actuels et futurs. Quelques redondances existent entre les différents tests benches mais ces codes sont entiers. Les composants testés, développés à COMELEC, n'ont délibérément pas été ajoutés à cette annexe.

i) test bench exhaustif VHDL puis "bdf"

```

-----
-- MULTICORE
-- replicate the test bench 2^replications times (replications=2 => 4cores)
-- area used limited to 32cores on our FPGA (max number of replications = 5)
-- max freq = 71 MHz with k=8, w=18 and 2 replications
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;

ENTITY MULTICORE IS

    generic(k : integer range 2 to 64 := 8;           -- size of the adders inputs
           w : integer range 2 to 64 := 18;         -- size of the error vector
           log2_w : integer range 1 to 6 := 5;      -- log2_w = ceil(log2(w))
           replications : integer range 0 to 5 := 0); -- total number of test benches replications

    port (clk, rst_n : in std_logic;
          Read_Addr : in UNSIGNED(log2_w-1 downto 0);

          dwn_cpt : out UNSIGNED(7 downto 0);
          COEF : out UNSIGNED(2*log2_w-1 downto 0);
          READ_INcN : out std_logic);

END ENTITY MULTICORE;

architecture comport of MULTICORE is
    -- internal signals descriptions:
    type dwn_cpt_matrix is array(integer range 0 to 2**replications-1) of UNSIGNED(7 downto 0);
    signal dwn_cpt_int : dwn_cpt_matrix;

    type COEF_matrix is array(integer range 0 to 2**replications-1) of UNSIGNED(w+4 downto 0);
    signal COEF_int, accu : COEF_matrix;

    type READ_INcN_matrix is array(integer range 0 to 2**replications-1) of std_logic;
    signal READ_INcN_int : READ_INcN_matrix;
begin

    test_cores : for i in 0 to 2**replications-1 generate
        test_core : entity work.TEST_BENCH_8bits_adder_rc(comport)
            generic map (k, w, log2_w, replications, i)
            -- if replications = 2 that generates 4 cores numbered i = 0 to 3.
            port map (clk, rst_n, READ_INcN_int(2**replications-1), Read_Addr, --inputs
                    dwn_cpt_int(i), COEF_int(i), READ_INcN_int(i)); --outputs
        end generate test_cores;

    --we only use the last core for the control:
    dwn_cpt <= dwn_cpt_int(2**replications-1);
    READ_INcN <= READ_INcN_int(2**replications-1);

    --add all respective COEF together from all cores:
    accu(0) <= COEF_int(0);
    cascade : for i in 1 to 2**replications-1 generate
        accu(i) <= COEF_int(i) + accu(i-1);
    end generate cascade ;

    --...and set the output to the result value:
    COEF <= CONV_UNSIGNED(accu(2**replications-1), COEF'length);

End architecture comport;
-----

```

```

-- TEST_BENCH_8bits_adder_rc

-- generic but limited in frequency and space used.
-- max freq = 74.09 MHz with k=8 and w=18
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all;
use ieee.std_logic_arith.ALL;

ENTITY TEST_BENCH_8bits_adder_rc IS

    generic(k : integer;           -- size of the adders inputs
            w : integer;           -- size of the error vector
            log2_w : integer;      -- log2_w = ceil(log2(w))
            replications : integer; -- total number of test benches replications
            core_nb : integer;     -- number of the test bench among the total

    port (clk, rst_n, READ_INcN GLOBAL : in std_logic;
          Read_Addr : in UNSIGNED(log2_w-1 downto 0);

          dwn_cpt : out UNSIGNED(7 downto 0);
          COEF : out UNSIGNED(w+4 downto 0);
          READ_INcN : out std_logic);

END ENTITY TEST_BENCH_8bits_adder_rc;

architecture comport of TEST_BENCH_8bits_adder_rc is
    signal a, b : UNSIGNED(k-1 downto 0);
    signal s, s_FAULTY : std_logic_vector(k-1 downto 0);
    signal c_out, c_out_FAULTY : std_logic;

    signal fault : UNSIGNED(w-1 downto 0);

    signal address, address_int : UNSIGNED(log2_w-1 downto 0);

    signal equal : std_logic;
begin
    --*****
    -- tested components:

    fault_FREE_adder : entity work.add_rc_hcmos_8(netlist)
        port map (std_logic_vector(a),std_logic_vector(b) , s,c_out);

    faulty_adder : entity work.add_rc_hcmos_8_faulty(netlist)
        port map (std_logic_vector(fault),std_logic_vector(a),std_logic_vector(b), s_FAULTY,c_out_FAULTY);

    --*****
    -- testing components:

    global_counter : entity work.counter(comport)
        generic map (k, w, replications, core_nb)
        -- example : 2 replications => 4 cores (nb 0 to 3)
        port map (clk,rst_n,READ_INcN_GLOBAL , a,b,fault,READ_INcN);

    parallel_adder : entity work.adress_generator(comport)
        generic map (w,log2_w)
        port map (fault,address);

    coef_accumulator : entity work.RAM_increm(comport)
        generic map (w, log2_w)
        port map (clk,rst_n,READ_INcN_GLOBAL,equal,address_int , COEF);

    --*****
    -- internal assignments:
    address_int <= Read_Addr when (READ_INcN_GLOBAL = '1')
        else address;

    equal <= '1' when (c_out_FAULTY & s_FAULTY = c_out & s )
        else '0';

    -- output assignement:
    dwn_cpt <= fault(w-1 downto w-8);

End architecture comport;

```

```

-----
-- counter

-- it's actually 3 nested counters.
-- it stops counting after an overflow
-- max freq: 312.50 MHz with k=8 and w=18 (but 53.78 MHz with k=32 and w=256)
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_arith.ALL;

ENTITY counter IS
  generic(k : integer range 2 to 64 := 8;           -- size of the adders inputs
         w : integer range 2 to 64 := 18;           -- size of the error vector
         replications : integer range 0 to 5 := 0;  -- total number of test benches replications
         core_nb : integer range 0 to 2**5 := 0);   -- number of the test bench among the total
  port (clk, rst_n, READ_INcN GLOBAL : in std_logic;
        cnt_a, cnt_b : out UNSIGNED(k-1 downto 0);
        err_vect : out UNSIGNED(w-1 downto 0);
        overflow : out std_logic);
END ENTITY counter;

architecture comport of counter is
  signal int_cnt : unsigned(2*k+w downto 0);
begin

  -- outputs affectations:
  cnt_a    <= int_cnt(k-1 downto 0);
  cnt_b    <= int_cnt(2*k-1 downto k);
  err_vect <= int_cnt(2*k+w-1 downto 2*k);
  overflow <= int_cnt(2*k+w);           -- it takes less area to test 1 bit than several

  process(clk, rst_n) is begin
    if (rst_n='0') then
      int_cnt <= (others => '0');
      int_cnt(2*k+w-1 downto 2*k+w-replications) <= conv_unsigned(core_nb , replications);
    elsif rising_edge(clk) then
      if (READ_INcN GLOBAL = '0') then
        int_cnt <= int_cnt + 1;         -- stop counting as soon as MSB = 1 (overflow)
      end if;
    end if;
  end process;
End architecture comport;

```

```

-----
-- adress generator

-- computes the hamming weight of the error vector input and deliver it in output
-- uses generic cascaded one bit adders (no pipeline, no tree)
-- worst propagation time = 13.819 ns (@50MHz limit = 256 inputs: Tp = 19.958 ns)
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all;
--USE ieee.std_logic_unsigned.ALL;
use ieee.std_logic_arith.ALL;

ENTITY adress_generator IS
  generic(w : integer range 2 to 64 := 18;           -- size of the error vector
         log2_w : integer range 1 to 6 := 5);       -- log2_w = ceil(log2(w))
  port (err_vec : in UNSIGNED(w-1 downto 0);
        address : out UNSIGNED(log2_w-1 downto 0));
END ENTITY adress_generator;

architecture comport of adress_generator is
  subtype vect is UNSIGNED(log2_w-1 downto 0);
  type matrix is ARRAY(integer range 0 to w-1) of vect;
  signal accu : matrix;
begin

  accu(0) <= (0=>err_vec(0) , others=>'0');

  cascade : for i in 1 to w-1 generate
    accu(i) <= err_vec(i) + accu(i-1);
  end generate cascade ;

  address <= accu(w-1);
End architecture comport;

```

```

-----
-- RAM increm
-- allows incrementing memory cells adressed by the input called nb_err
-- MAX FREQ = 191.79 MHz with w=18
-----

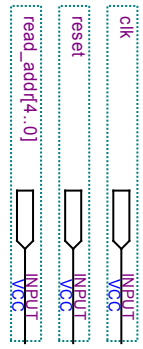
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all; -- use functions LOG() and
CEIL()
--USE ieee.std_logic_unsigned.ALL;
use ieee.std_logic_arith.ALL;

ENTITY RAM_increm IS
    generic(w : integer range 2 to 64 := 18; -- size of the error vector
           log2_w : integer range 1 to 6 := 5); -- log2_w = log2(w)
    port (ck, reset, READ_INcN_GLOBAL, equal : in std_logic;
          address : in UNSIGNED(log2_w-1 downto 0);
          COEF : out UNSIGNED(w+4 downto 0));
END ENTITY RAM_increm;

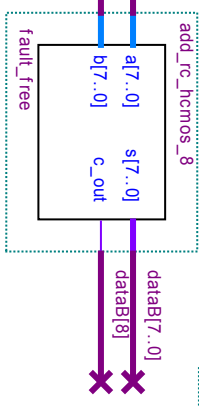
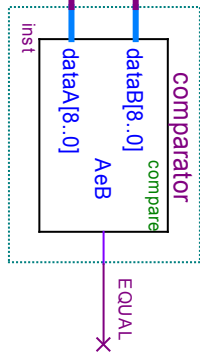
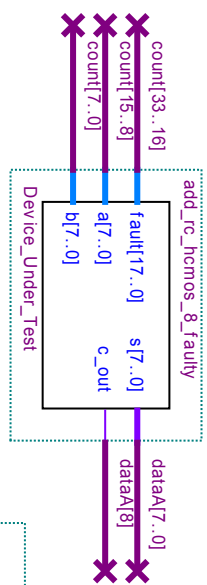
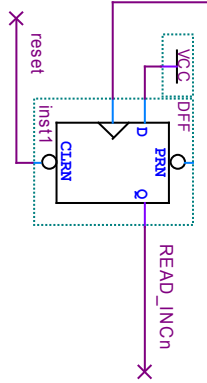
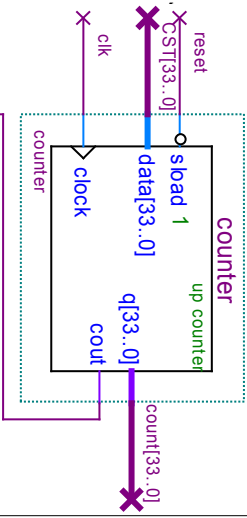
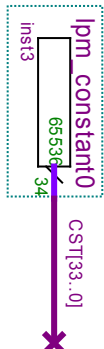
architecture comport of RAM_increm is
    type RAM is ARRAY(integer range 0 to w) of UNSIGNED(w+4 downto 0);
    signal cell : RAM;
begin
    process(ck, reset) is begin
        if (reset='0') then
            cell <= (others=>(others=>'0'));
        elsif rising_edge(ck) then
            if (READ_INcN_GLOBAL = '0') then -- INcReMent mode
                if (equal = '1' AND conv_integer(address) <= w) then
                    cell(conv_integer(address)) <= cell(conv_integer(address)) + 1;
                end if;
            else -- ReaD mode
                COEF <= cell(conv_integer(address));
            end if;
        end if;
    end process;
End architecture comport;

```

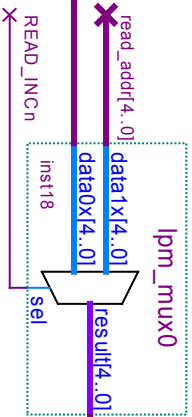
Avant d'avoir effectué cette longue description en VHDL générique, une description graphique a été faite pour valider le procédé. Un aperçu de cette description graphique est en page suivante.



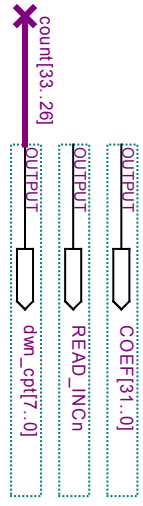
load '1' in q[16] when reset=0



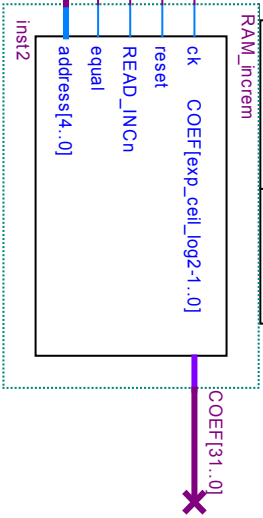
Parameter	Value
PIPELINE	
REPRESENTATION	"UNSIGNED"
SHIFT	
SIZE	18
WIDTH	1
WIDTHR	5



WORST PROP. TIME = 18ns



Parameter	Value
N	18
EXP_CELL_LOG2	32



ii) test bench aléatoire VHDL

```

-----
-- TEST_BENCH_8bits_adder_rc RANDOM GENERATING VERSION
-- generic but limited in frequency and space used.
-- only calculating for 4 coefficients
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;

ENTITY TEST_BENCH_8bits_adder_rc IS
    generic(k : positive range 2 to 64 := 8;           -- size of the adders inputs
           w : positive range 2 to 64 := 18;         -- size of the error vector
           log2_w : positive range 1 to 6 := 5;      -- log2_w = ceil(log2(w))
           loop_size : positive := 20);             -- 2**loop_size = nb of loops of the random test

    port (clk, rst_n, unfreeze_n : in std_logic;
          Read_Addr : in std_logic_vector(1 downto 0); --just 2 bits for just 4 coefficients
          dwn_cpt : out std_logic_vector(7 downto 0);
          COEF : out std_logic_vector(31 downto 0); --31=2**log2_w-1
          READ_INcN : out std_logic);
END ENTITY TEST_BENCH_8bits_adder_rc;

architecture comport of TEST_BENCH_8bits_adder_rc is
    signal a, b : std_logic_vector(k-1 downto 0);
    signal s, s_FAULTY : std_logic_vector(k-1 downto 0);
    signal c_out, c_out_FAULTY : std_logic;
    signal fault : std_logic_vector(w-1 downto 0);
    signal overflow : std_logic;
    signal address, address_int : std_logic_vector(1 downto 0);
    signal equal, cnt_enable, sames : std_logic;
    signal error_vector : std_logic_vector(2**log2_w-1 downto 0);
    constant zeros : std_logic_vector(2**log2_w-1 downto w) := (others=>'0');
begin
    --*****
    -- testing components:

    RANDOM_GEN_A : entity work.PRG(comport)
        generic map (k, "1010101") port map (clk,rst_n , a);

    RANDOM_GEN_B : entity work.PRG(comport)
        generic map (k, "1010001") port map (clk,rst_n , b);

    FAULT_GEN : entity work.EVG(comport)
        generic map (log2_w) port map (clk,rst_n,address , error_vector,sames);

        --get the LSB of generated faults...
        fault <= error_vector(w-1 downto 0);

        --allows to disable the counter when injected errors are out of the fault bus width
        cnt_enable <= '1' when (error_vector(2**log2_w-1 downto w)=zeros AND (sames='0' OR unfreeze_n='0'))
            else '0';

    loop_controller : entity work.counter(comport)
        generic map (loop_size) port map (clk,rst_n,cnt_enable , address,overflow);

    coef_accumulator : entity work.RAM_increm(comport)
        port map (clk,rst_n,overflow,equal,address_int , COEF);

        address_int <= Read_Addr when (overflow = '1')
            else address;

        -- increment when masking detected, only if generated faults are not too long:
        equal <= '1' when ((c_out_FAULTY & s_FAULTY) = (c_out & s) AND cnt_enable = '1')
            else '0';

    --*****
    -- tested components:

    fault_FREE_adder : entity work.add_rc_hcmos_8(netlist)
        port map (std_logic_vector(a),std_logic_vector(b) , s,c_out);

    faulty_adder : entity work.add_rc_hcmos_8_faulty(netlist)
        port map (std_logic_vector(fault),std_logic_vector(a),std_logic_vector(b) , s_FAULTY,c_out_FAULTY);

    --*****
    -- outputs assignments:

    READ_INcN <= overflow;

    dwn_cpt <= "00000" & overflow & address;

End architecture comport;
-----

```

```

-- counter

-- stops counting after an overflow
-- count disabled when generated errors occur out of the fault bus width
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;

ENTITY counter IS
    generic(size_cpt : positive range 2 to 64 := 20);           -- size of the "counter" for statistics iterations
    port (clk, rst_n, cnt_enable : in std_logic;
          nb_err : out std_logic_vector(1 downto 0);           -- allow test for 1 to 4 errors.
          overflow : out std_logic);
END ENTITY counter;

architecture comport of counter is
    signal int_cnt : UNSIGNED(size_cpt+3 downto 0);
begin

    -- outputs affectations:
    nb_err <= std_logic_vector(int_cnt(size_cpt+2 downto size_cpt+1));
    overflow <= int_cnt(size_cpt+3);

    process(clk, rst_n) is begin
        if (rst_n='0') then
            int_cnt <= (others=>'0');
        elsif rising_edge(clk) then
            if (int_cnt(size_cpt+3)='0' AND cnt_enable='1') then -- count untill MSB = 1 (overflow)
                int_cnt <= int_cnt + 1;                          -- stops counting if disabled by input
            end if;
        end if;
    end process;

End architecture comport;

```

```

-----
-- RAM_increm

-- allows incrementing memory cells adressed by the input called nb_err
-- the memory cell at the adress 0 is not used because we don't count 0 fault cases
-- the overflow of a memory cell would stop its incrementation (then result = 0)
-- MAX_FREQ = 191.79 Mhz with w=18
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;                                     -- for the type conversions
USE ieee.std_logic_unsigned.ALL;                             -- to use the "+" OPERATOR

ENTITY RAM_increm IS
    generic(COEF_SIZE : positive range 2 to 64 := 32);        -- size of COEF
    port (ck, reset, READ_INcN, equal : in std_logic;
          address : in std_logic_vector(1 downto 0);
          COEF : out std_logic_vector(COEF_SIZE-1 downto 0));
END ENTITY RAM_increm;

architecture comport of RAM_increm is
    type RAM is ARRAY(integer range 0 to 3) of std_logic_vector(COEF_SIZE downto 0);
    signal cell : RAM;
begin

    process(ck, reset) is begin
        if (reset='0') then
            cell <= (others=>(others=>'0'));
        elsif rising_edge(ck) then

            if (READ_INcN = '0') then                            -- INCrement mode, only when no overflow (when MSB=0)
                if (equal = '1') then
                    if (cell(to_integer(unsigned(address)))(COEF_SIZE) = '0') then
                        cell(to_integer(unsigned(address))) <= cell(to_integer(unsigned(address))) + 1;
                    else -- if overflow then mark it with 1&333333333 like "3rror":
                        cell(to_integer(unsigned(address))) <= X"1C6AEA155";
                    end if;
                end if;
            else                                                -- ReaD mode
                COEF <= cell(to_integer(unsigned(address)))(COEF_SIZE-1 downto 0);
            end if;
        end if;
    end process;

End architecture comport;

```

```

-----
-- Error Vector Generator: (output on log2 w bits)

-- set @ 1 the bits for of which adresses are randomly generated by the PRG (see above)
-- note: hamming weight of error_vector = switch_err + 1 = [1;4]
-- use: just change the log2_w value in the 1st generic statement
-- (the log2_w is transmitted to the PRG by the bdf instantiation)
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

ENTITY EVG IS
  generic(log2_w : positive range 3 to 32 := 5); -- default: log2_w = 5 to get "random" vectors on 32 bits
  port (ck, r : in std_logic;
        switch_err : in std_logic_vector(1 downto 0);
        fault : out std_logic_vector((2**log2_w)-1 downto 0);
        sames : out std_logic);
END ENTITY EVG;

architecture comport of EVG is
  signal s_addr0, s_addr1, s_addr2, s_addr3 : std_logic_vector(log2_w-1 downto 0);
  signal error_vector: std_logic_vector((2**log2_w)-1 downto 0);
begin
  -- Pseudo Random Generator (PRG) is described after this component ( used to get random adresses)
  PRG0 : entity work.PRg(comport)
    generic map (log2_w, "1010110") port map (ck, r, s_addr0);
  PRG1 : entity work.PRg(comport)
    generic map (log2_w, "1011101") port map (ck, r, s_addr1);
  PRG2 : entity work.PRg(comport)
    generic map (log2_w, "1000100") port map (ck, r, s_addr2);
  PRG3 : entity work.PRg(comport)
    generic map (log2_w, "1110010") port map (ck, r, s_addr3);

  fault <= error_vector;

  process(ck,r) is begin
    if (r='0') then
      error_vector <= (others => '0');
      sames <= '0';

    else
      case switch_err is
        when "00" => --hamming weight = 1
          error_vector <= (others => '0');
          error_vector(to_integer(unsigned(s_addr0))) <= '1';
          sames <= '0';

        when "01" => --hamming weight = 2
          error_vector <= (others => '0');
          error_vector(to_integer(unsigned(s_addr0))) <= '1';
          error_vector(to_integer(unsigned(s_addr1))) <= '1';

          if (s_addr0=s_addr1) then
            sames <= '1';
          else
            sames <= '0';
          end if;

        when "10" => --hamming weight = 3
          error_vector <= (others => '0');
          error_vector(to_integer(unsigned(s_addr0))) <= '1';
          error_vector(to_integer(unsigned(s_addr1))) <= '1';
          error_vector(to_integer(unsigned(s_addr2))) <= '1';

          if (s_addr0=s_addr1 OR s_addr0=s_addr2 OR s_addr2=s_addr1) then
            sames <= '1';
          else
            sames <= '0';
          end if;

        when "11" => --hamming weight = 4
          error_vector <= (others => '0');
          error_vector(to_integer(unsigned(s_addr0))) <= '1';
          error_vector(to_integer(unsigned(s_addr1))) <= '1';
          error_vector(to_integer(unsigned(s_addr2))) <= '1';
          error_vector(to_integer(unsigned(s_addr3))) <= '1';

          if (s_addr0=s_addr1 OR s_addr0=s_addr2 OR s_addr0=s_addr3 OR s_addr1=s_addr2
              OR s_addr1=s_addr3 OR s_addr2=s_addr3) then
            sames <= '1';
          else
            sames <= '0';
          end if;
      end case;
    end if;
  end process;
End architecture comport;

```

```

-----
-- Pseudo Random Generator: (output on log2 w bits)
-- Uses a Linear feedback shift register to generate random addresses for the EVG
-- The internal reg uses 4 more bits for more randomness (16 times more)
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all; -- needed to use functions LOG() and CEIL()
--USE ieee.std_logic_unsigned.ALL;
--use ieee.std_logic_arith.ALL;

ENTITY PRG IS
    generic (log2_w : positive range 3 to 32 := 3; -- default: log2_w = 3 to get "random" values on 3 bits
            seed : unsigned(6 downto 0) := "0000010"); -- allow to start with different values
    port (clk,reset : in std_logic;
          rand : out std_logic_vector(log2_w-1 downto 0));
END ENTITY PRG;

architecture comport of PRG is
    signal reg : unsigned(log2_w+3 downto 0);
begin
    process(clk,reset) is begin

        if (reset='0') then
            reg <= resize(seed, reg'length) OR reg; -- reuse the previous reg value (if several tests)
        elsif rising_edge(clk) then
            reg(log2_w+3 downto 1) <= reg(log2_w+2 downto 0); -- shift register function
            reg(0) <= reg(log2_w+3) xor reg(log2_w+2); -- random generation function
        end if;
    end process;

    rand <= std_logic_vector(reg(log2_w-1 downto 0)); -- get the log2_w bits useful in output
END ARCHITECTURE comport;

```

iii) test bench 1 erreur VHDL

```

--*****
--*** systematic version :
--*** to change the name of the tested components look in the component called TEST_BENCH ***
--*****

-----
-- MULTICORE FOR 1 ERROR INJECTED ONLY !!!

-- replicate the test bench 2^replications times
-- area used limited to 32cores on our FPGA (max number of replications = 5)
-- max freq = 71 MHz with k=8, w=18 and 2 replications (replications=2 => 4cores)
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all;
use ieee.std_logic_arith.ALL;

ENTITY MULTICORE IS

    generic(k : integer range 2 to 16 := 8;           -- size of the components inputs
            w : integer range 2 to 256 := 18;        -- size of the error vector
            log2_w : integer range 1 to 8 := 5;      -- log2_w = ceil(log2(w))
            replications : integer range 0 to 5 := 0); -- total number of test benches replications

    port (clk, rst_n : in std_logic;
          Read_Addr : in UNSIGNED(7 downto 0);      -- temporary :8 bits (allows 256 choices if w=256)

          dwn_cpt : out UNSIGNED(7 downto 0);
          COEF : out UNSIGNED(31 downto 0):=(others=>'0'); -- max(COEF)=2^(2*k-1) If any error is masked
          READ_INCN : out std_logic);

END ENTITY MULTICORE;

architecture comport of MULTICORE is
    -- internal signals descriptions:
    type dwn_cpt_matrix is array(integer range 0 to 2**replications-1) of UNSIGNED(7 downto 0);
    signal dwn_cpt_int : dwn_cpt_matrix;

    type COEF_matrix is array(integer range 0 to 2**replications-1) of UNSIGNED(2*k-1 downto 0);
    signal COEF_int, accu : COEF_matrix;

    type READ_INCN_matrix is array(integer range 0 to 2**replications-1) of std_logic;
    signal READ_INCN_int : READ_INCN_matrix;
begin
    test_cores : for i in 0 to 2**replications-1 generate
        test_core : entity work.TEST_BENCH(comport)
            generic map (k, w, log2_w, replications, i)
            -- if replications = 2 that generates 4 cores numbered i = 0 to 3.
            port map (clk, rst_n, READ_INCN_int(2**replications-1), Read_Addr(log2_w-1 downto 0), --input
                    dwn_cpt_int(i), COEF_int(i), READ_INCN_int(i)); --output
    end generate test_cores;

    --we only use the last core for the control:
    dwn_cpt <= dwn_cpt_int(2**replications-1);
    READ_INCN <= READ_INCN_int(2**replications-1);

    --add all respective COEF together from all cores:
    accu(0) <= COEF_int(0);
    cascade : for i in 1 to 2**replications-1 generate
        accu(i) <= COEF_int(i) + accu(i-1);
    end generate cascade ;

    --...and set the output to the result value:
    COEF(2*k-1 downto 0) <= accu(2**replications-1);

End architecture comport;

```

```

-----
-- TEST_BENCH

-- generic but limited in frequency and space used.
-- max freq = 74.09 MHz with k=8 and w=18
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
--use IEEE.NUMERIC_STD.ALL;
--USE ieee.math_real.all;
use ieee.std_logic_arith.ALL;

ENTITY TEST_BENCH IS

    generic(k : integer;           -- size of the components inputs
           w : integer;           -- size of the error vector
           log2_w : integer;      -- log2_w = ceil(log2(w))
           replications : integer; -- total number of test benches replications
           core_nb : integer);    -- number of the test bench among the total

    port (clk, rst_n, READ_INcN_GLOBAL : in std_logic;
          Read_Addr : in UNSIGNED(log2_w-1 downto 0);

          dwn_cpt : out UNSIGNED(7 downto 0);
          COEF : out UNSIGNED(2*k-1 downto 0);
          READ_INcN : out std_logic);

END ENTITY TEST_BENCH;

architecture comport of TEST_BENCH is
    signal a, b : UNSIGNED(k-1 downto 0);
    signal s, s_FAULTY : std_logic_vector(k-1 downto 0);
    signal c_out, c_out_FAULTY : std_logic_vector(1 downto 0);
    signal fault : UNSIGNED(w-1 downto 0);
    signal address, address_int : UNSIGNED(log2_w-1 downto 0);
    signal equal : std_logic;
begin

    --*****
    --***** tested components: *****
    --*****

    fault_FREE_component : entity work.add_rc_hcmos_8(netlist)
    port map (std_logic_vector(a),std_logic_vector(b) , s,c_out);

    faulty_component : entity work.add_rc_hcmos_8_faulty(netlist)
    port map (std_logic_vector(fault),std_logic_vector(a),std_logic_vector(b), s_FAULTY,c_out_FAULTY);

    --*****
    --*****

    -- testing components:

    global_counter : entity work.counter(comport)
    generic map (k, w, log2_w, replications, core_nb)
    -- example : 2 replications => 4 cores 0 to 3
    port map (clk,rst_n,READ_INcN_GLOBAL , a,b,fault,address,READ_INcN);

    coef_accumulator : entity work.RAM_increm(comport)
    generic map (k, w, log2_w)
    port map (clk,rst_n,READ_INcN_GLOBAL,equal,address_int , COEF);

    -- internal assignments:
    address_int <= Read_Addr when (READ_INcN_GLOBAL = '1')
    else address;

    equal <= '1' when (c_out_FAULTY & s_FAULTY = c_out & s)
    else '0';

    -- output assignement:
    ok : if k>=8 generate
    dwn_cpt <= b(k-1 downto k-8);
    end generate ok;

    small : if k=4 generate
    dwn_cpt <= b&a;
    end generate small;

End architecture comport;

```

```

-----
-- counter
-- the error vector starts with the value 1 to avoid the 0 error case.
-- it stops counting after an overflow -- max freq: 425 MHz with k=16 and w=256
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;

ENTITY counter IS
  generic(k : integer := 8; -- size of the components inputs
         w : integer := 18; -- size of the error vector
         log2_w : integer := 5; -- log2_w = ceil(log2(w))
         replications : integer := 0; -- total number of test benches replications
         core_nb : integer := 0); -- number of the test bench among the total
  port (clk, rst_n, READ_INcN_GLOBAL : in std_logic;
        cnt_a, cnt_b : out UNSIGNED(k-1 downto 0);
        err_vect : out UNSIGNED(w-1 downto 0);
        address : out UNSIGNED(log2_w-1 downto 0);
        overflow : out std_logic);
END ENTITY counter;

architecture comport of counter is
  signal int_cnt : unsigned(2*k downto 0);
  signal rotate_reg : unsigned(w-1 downto 0);
  signal temp : UNSIGNED(log2_w-1 downto 0);
begin
  -- outputs affectations:
  address <= temp;
  err_vect <= rotate_reg;
  cnt_a <= int_cnt(k-1 downto 0);
  cnt_b <= int_cnt(2*k-1 downto k);
  overflow <= int_cnt(2*k);

  process(clk, rst_n) is begin
    if (rst_n='0') then
      int_cnt <= (others => '0');
      int_cnt(2*k-1 downto 2*k-replications) <= conv_unsigned(core_nb , replications);
      rotate_reg <= (0=>'1', others=>'0'); --initialise the rotate register at 1
      temp <= (others => '0');
    elsif rising_edge(clk) then
      if (READ_INcN_GLOBAL = '0') then --rotate left
        rotate_reg <= rotate_reg(w-2 downto 0) & rotate_reg(w-1);
        if (rotate_reg(w-1) = '0') then --check if the rotation is not finished:
          temp <= temp + 1; -- will be used to adress the correct memory cell
        else --at each rotation of the rotate register:
          int_cnt <= int_cnt + 1; -- increment the inputs injection counter
          temp <= (others => '0'); -- reset the address counter
        end if;
      end if;
    end if;
  end process;
End architecture comport;

```

```

-----
-- RAM_increm
-- allows incrementing memory cells addressed by the input called nb err
-- the memory cell at the address 0 is not used because we don't count 0 fault cases
-- MAX_FREQ = 191.79 MHz with w=18
-----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;

ENTITY RAM_increm IS
  generic(k : integer := 8; -- size of the components inputs
         w : integer := 18; -- size of the error vector
         log2_w : integer := 5); -- log2_w = log2(w)
  port (ck, reset, READ_INcN_GLOBAL, equal : in std_logic;
        address : in UNSIGNED(log2_w-1 downto 0);
        COEF : out UNSIGNED(2*k-1 downto 0));
END ENTITY RAM_increm;

architecture comport of RAM_increm is
  type RAM_COEF is ARRAY(integer range 0 to w-1) of UNSIGNED(2*k-1 downto 0);
  signal cell : RAM_COEF;
begin
  process(ck, reset) is begin
    if (reset='0') then
      cell <= (others=>(others=>'0'));
    elsif rising_edge(ck) then
      if (READ_INcN_GLOBAL = '0') then -- INcReMent mode
        if (equal = '1' AND conv_integer(address) < w) then
          cell(conv_integer(address)) <= cell(conv_integer(address)) + 1;
        end if;
      else -- ReaD mode
        COEF <= cell(conv_integer(address));
      end if;
    end if;
  end process;
End architecture comport;

```

iv) code C embarqué

```

//define the number of coefficients to get back : (= nb of bits in error vector)
#define coef_nb 18 // case of the RC adder 8bits

#include "count_binary.h"
#include <time.h>
#include <stdio.h>

// A variable to hold the value of the button pio edge capture register.
volatile int edge_capture;
// used to read the computation state:
static alt_u8 ready;
// used to show the computation state:
static alt_u8 dwn_cpt;
// coefficient calculated by our test bench
static alt_u32 coef;
// adress used to request coefficient from test bench
static alt_u8 addr_out;
// used to restart the test:
static alt_u8 restart;
// used to count the calculation time:
static clock_t start;
static double the_time;

static void handle_button_interrupts(void* context, alt_u32 id)
{
    /* Cast context to edge_capture's type. It is important that this be
    * declared volatile to avoid unwanted compiler optimization.*/
    volatile int* edge_capture_ptr = (volatile int*) context;
    /* Store the value in the Button's edge capture register in *context. */
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
    /* Reset the Button's edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
}

/* Initialize the button pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the alt_irq_register() function
    * prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;
    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    /* Register the interrupt handler. */
    alt_irq_register( BUTTON_PIO_IRQ, edge_capture_ptr,
        handle_button_interrupts );
}

/* Seven Segment Display PIO Functions */
static void count_sevenseg(int hex)
{
    static alt_u8 segments[16] = {
        0x81, 0xcf, 0x92, 0x86, 0xcc, 0xa4, 0xa0, 0x8f, 0x80, 0x84, /* 0-9 */
        0x88, 0xe0, 0xf2, 0xc2, 0xb0, 0xb8 }; /* a-f */

    unsigned int data = segments[hex & 15] | (segments[(hex >> 4) & 15] << 8);

    IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_PIO_BASE, data);
}

/* Illuminate LEDs with the argument given */
static void count_led(alt_u8 b)
{
    /* make the LED's LSB on the right (= bitwise swap)
    IOWR_ALTERA_AVALON_PIO_DATA
    (
        LED_PIO_BASE,
        ( (b * 0x0802LU & 0x22110LU) | (b * 0x8020LU & 0x88440LU) ) * 0x10101LU >> 16
    );
}

static void led_sevenseg(alt_u8 hex)
{
    count_sevenseg(hex);
    count_led(hex);
}

static void refresh_IO()
{
    /* refresh "addr_out" to send the good coef request :
    IOWR_ALTERA_AVALON_PIO_DATA( ADDRESS_COEF_BASE , addr_out); // ADDRESS_COEF_BASE = output 5bits to RAM

    /* GET THE COMPUTATION STATE (finish or not):
    ready = IORD_ALTERA_AVALON_PIO_DATA( READY_BASE ); // READY_BASE = input 1bit

    /* GET THE COMPUTATION "TIME" UNTILL THE END:
    dwn_cpt = IORD_ALTERA_AVALON_PIO_DATA( DOWN_CPT_BASE ); // DOWN_CPT_BASE = input 8bit

    /* Get the coefficient from the RAM :
    coef = IORD_ALTERA_AVALON_PIO_DATA( COEF_BASE ); // COEF_BASE = input 32bits from RAM
}

static void reset_TEST_BENCH()
{
    IOWR_ALTERA_AVALON_PIO_DATA( RESET_TEST_BENCH_BASE , 0); //reset
    usleep(10); //wait 10 µs = 500Tclk
    IOWR_ALTERA_AVALON_PIO_DATA( RESET_TEST_BENCH_BASE , 1); //start
}

```

```

static void handle_button_press()
{
    switch(edge_capture)
    {
        case 0x1: //button 0 : start another iteration
            reset_TEST_BENCH();
            break;

        default:
            printf("recommencez!!!\r\n");
            break;
    }
    edge_capture = 0; //reinit. interrup. flag.
}

static void init_main()
{
    refresh_IO();
    led_sevenseg(0);
    addr_out = 0;
    dwn_cpt=0xFF;
    printf("\r\n\r\n\r\nSTART...\r\n\r\n");
    start = clock(); //start stopwatch
}

/*****
 * int main()
 *****/
int main(void)
{
    //initialisations:
    init_button_pio();
    init_main();
    reset_TEST_BENCH();

    // Endless loop:
    while( 1 )
    {
        usleep(100000); //wait 0.1s

        if (ready==0) //show if calculation is not yet done:
        {
            refresh_IO();
            led_sevenseg(0xFF - dwn_cpt); //show when it's going to finish:
        }
        else //if calculation is finished then get results
        {
            while (addr_out<coef_nb)
            {
                if (addr_out == 0)
                {
                    //get calculation time:
                    the_time = (double)(clock()-start)/(double)(CLOCKS_PER_SEC);
                    //display "the_time" in elapsed seconds:
                    printf("...calculation time: %d min & %2.2f sec ( = %4.2f sec )\r\n\r\n",
                        (int)(the_time/60), (int)the_time%60 + the_time - (int)the_time, the_time);
                }

                refresh_IO();

                printf("ERROR_RANK(%d)=%u;\r\n", addr_out+1, coef);
                addr_out++;
            }
            printf("\r\n sum(ERROR_RANK)\r\n");

            // check for the reset request
            restart = 0;
            while( restart == 0 )
            {
                if (edge_capture!=0) //reset requested (push button interrupt)
                {
                    handle_button_press();
                    init_main();
                    restart = 1;
                }
                else usleep(1000000); //wait 1s
            }
        }
    }
    return 0;
}

/////////////////////////////////////////////////////////////////
// can be useful to send a digit from keyboard to FPGA: //
/////////////////////////////////////////////////////////////////
/*
static void saisie_clavier()
{
    unsigned int tmp;
    printf( "\r\n\r\n Entrez une valeur hexa à envoyer au FPGA : \r\n");
    scanf("%1x", &tmp);
    addr_out=(alt_u8)tmp;
    printf( "\r\n Vous avez saisi : %1x, merci \r\n", addr_out);
}
*/

```

Aperçu d'utilisation du code C embarqué:

The screenshot shows the Nios II IDE interface. The top menu bar includes File, Edit, Refactor, Navigate, Search, Project, Tools, Run, Window, and Help. The Navigator pane on the left lists project files, with 'main' selected. The main editor displays the source code for 'count_binary.c', which includes a main function with initializations and a while loop. The Console pane at the bottom shows the execution output, including a calculation time and a list of error ranks for 16 iterations. A red bracket groups the error rank list, and a red dashed box contains the text 'rappatriement des résultats calculés'.

```

# coef_nb
count_binary.h
time.h
stdio.h
edge_capture
ready
dwn_cpt
coef
addr_out
restart
start
the_time
handle_button_interrupts
init_button_pio
count_sevenseg
count_led
led_sevenseg
refresh_IO
reset_TEST_BENCH
handle_button_press
init_main
main

int main(void)
{
    //initialisations:
    init_button_pio();
    init_main();
    reset_TEST_BENCH();

    // Endless loop:
    while( 1 )
    {
        usleep(100000);           //w
        if (ready==0)           //s
        {
            refresh_IO();
            led_sevenseg(0xFF - dwn_cpt);
        }
        else                       //i
        {

```

```

<terminated> hello_led_0 Nios II HW configuration [Nios II Hardware] Nios II Download output (27/06/08 15:02)
[Console output redirected to file:C:\altera\stage\stdio.txt]

START...

...calculation time: 0 min and 0.19 sec

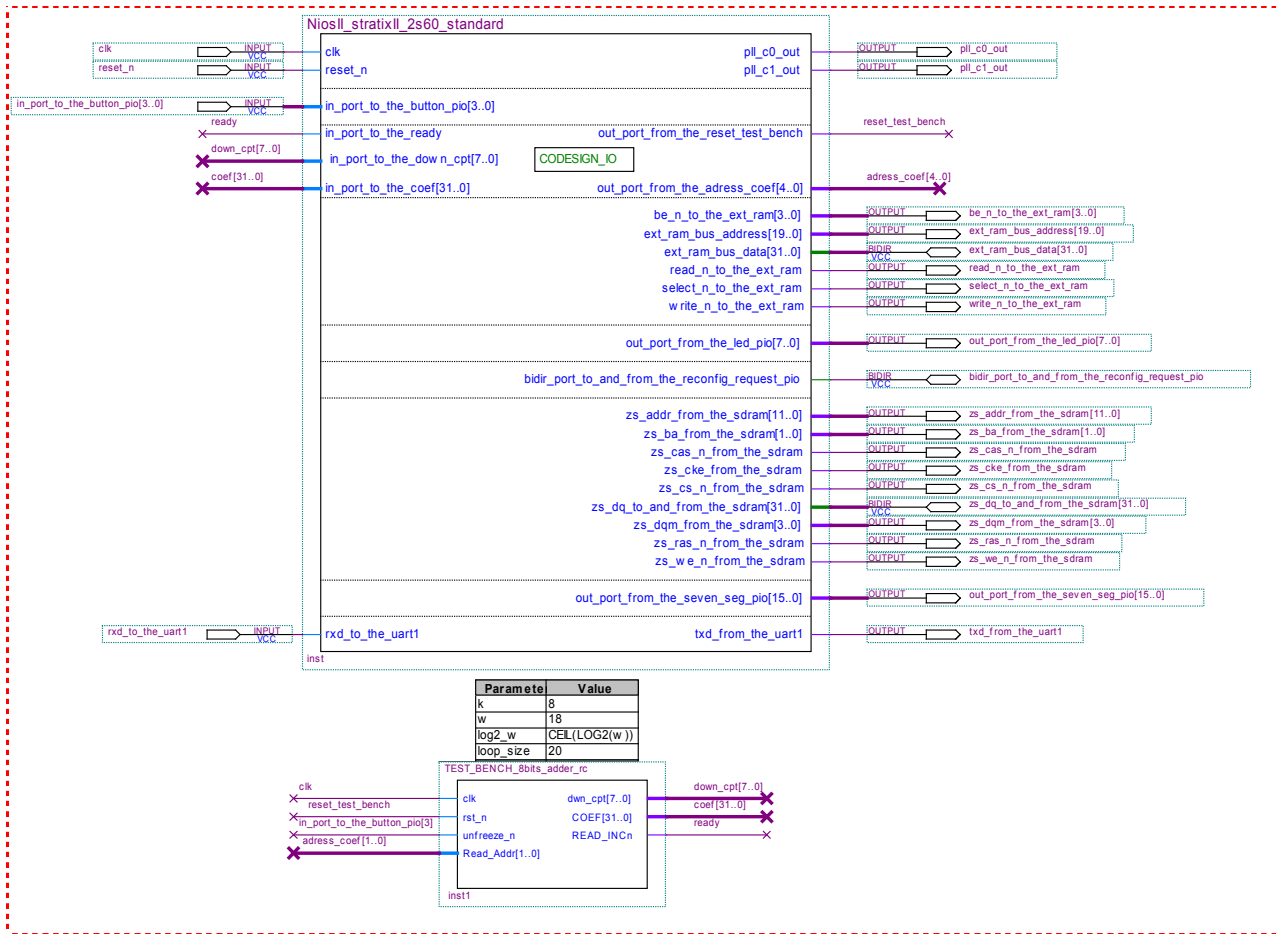
ERROR_RANK (1)=48;
ERROR_RANK (2)=0;
ERROR_RANK (3)=112;
ERROR_RANK (4)=0;
ERROR_RANK (5)=96;
ERROR_RANK (6)=96;
ERROR_RANK (7)=216;
ERROR_RANK (8)=224;
ERROR_RANK (9)=32;
ERROR_RANK (10)=216;
ERROR_RANK (11)=224;
ERROR_RANK (12)=208;
ERROR_RANK (13)=36;
ERROR_RANK (14)=112;
ERROR_RANK (15)=34;
ERROR_RANK (16)=112;

```

rappatriement des résultats calculés

b) Schémas de synthèse

Processeur NIOS II et test bench : description graphique



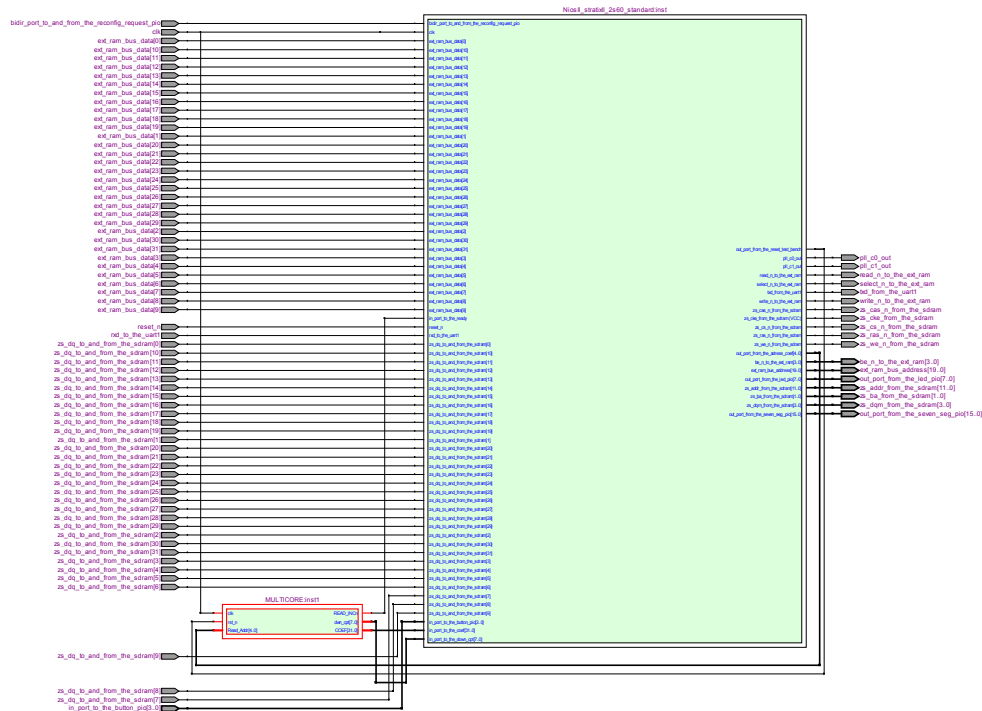
Processeur NIOS II et test bench : description dans SOPC builder:

Target
Device Family: Stratix II

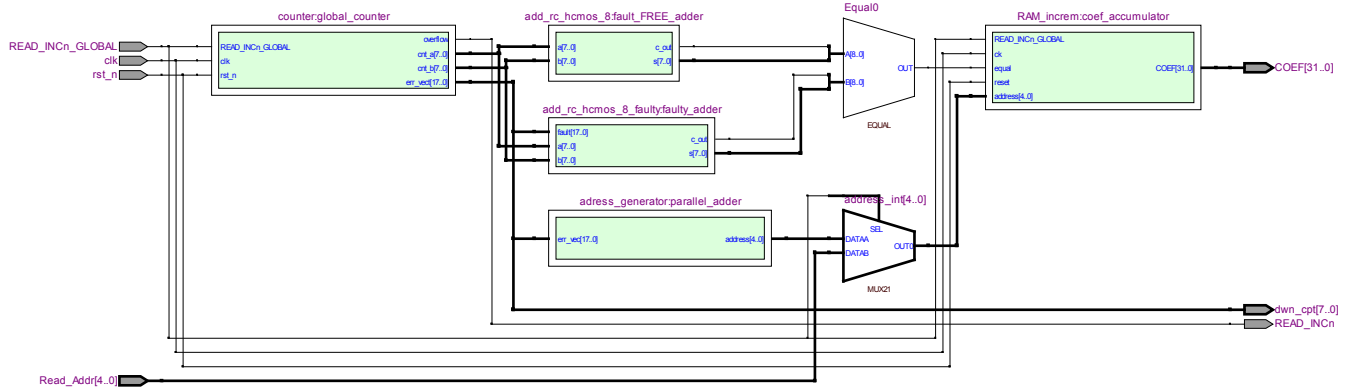
Name	Source	MHz
clk	External	50,0
pll_c0	pll.c0	50,0
pll_c1	pll.c1	50,0

Use	Co...	Module Name	Description	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		pll	PLL	clk	0x02221060	0x0222107f	
<input checked="" type="checkbox"/>		cpu	Nios II Processor	pll_c0	0x02220800	0x02220fff	
<input checked="" type="checkbox"/>		sys_clk_timer	Interval Timer	pll_c0	0x02221000	0x0222101f	
<input checked="" type="checkbox"/>		sysid	System ID Peripheral	pll_c0	0x02221110	0x02221117	
<input checked="" type="checkbox"/>		reconfig_request_pio	PIO (Parallel I/O)	pll_c0	0x02221080	0x0222108f	
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART	pll_c0	0x02221118	0x0222111f	
<input checked="" type="checkbox"/>		ext_ram_bus	Avalon-MM Tristate Bridge	pll_c0			
<input checked="" type="checkbox"/>		high_res_timer	Interval Timer	pll_c0	0x02221020	0x0222103f	
<input checked="" type="checkbox"/>		ext_ram	IDT71V416 SRAM	pll_c0	0x02100000	0x021fffff	
<input checked="" type="checkbox"/>		uart1	UART (RS-232 Serial Port)	pll_c0	0x02221040	0x0222105f	
<input checked="" type="checkbox"/>		button_pio	PIO (Parallel I/O)	pll_c0	0x02221090	0x0222109f	
<input checked="" type="checkbox"/>		led_pio	PIO (Parallel I/O)	pll_c0	0x022210a0	0x022210af	
<input checked="" type="checkbox"/>		seven_seg_pio	PIO (Parallel I/O)	pll_c0	0x022210b0	0x022210bf	
<input checked="" type="checkbox"/>		onchip_ram	On-Chip Memory (RAM or ROM)	pll_c0	0x02210000	0x0221ffff	
<input checked="" type="checkbox"/>		sdram	SDRAM Controller	pll_c0	0x01000000	0x01fffff	
<input checked="" type="checkbox"/>		ready	PIO (Parallel I/O)	pll_c0	0x022210c0	0x022210cf	
<input checked="" type="checkbox"/>		address_coef	PIO (Parallel I/O)	pll_c0	0x022210d0	0x022210df	
<input checked="" type="checkbox"/>		coef	PIO (Parallel I/O)	pll_c0	0x022210e0	0x022210ef	
<input checked="" type="checkbox"/>		reset_test_bench	PIO (Parallel I/O)	pll_c0	0x022210f0	0x022210ff	
<input checked="" type="checkbox"/>		down_cpt	PIO (Parallel I/O)	pll_c0	0x02221100	0x0222110f	

Processeur NIOS II et test bench : schéma de synthèse



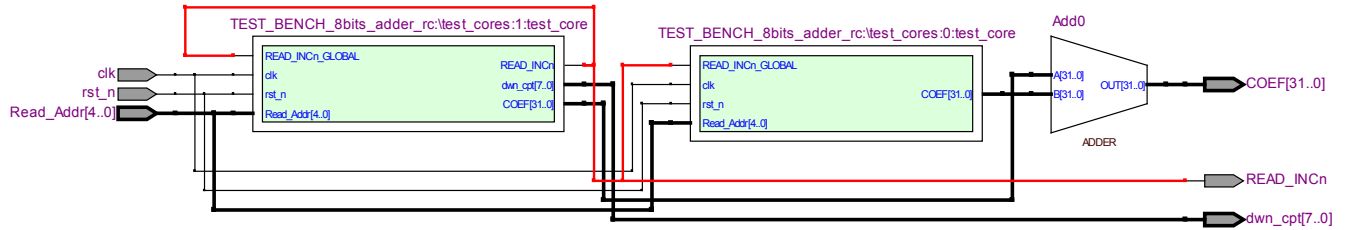
Test bench à "1 cœur": (temps de calcul: 343.74s - utilisations* ALUTs: 7% - registres: 6%)



Compilation: 9 min (avec le Nios)

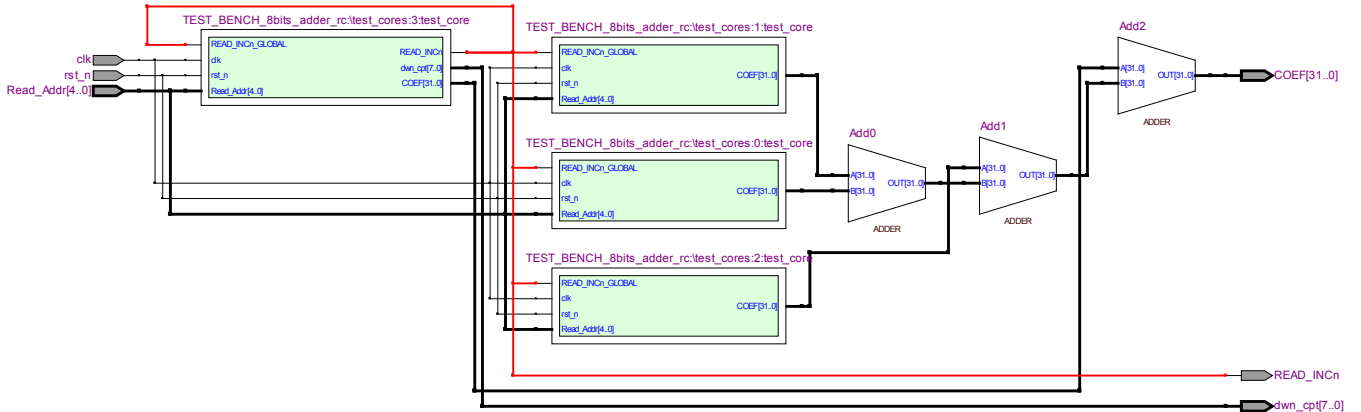
*Note: le FPGA utilisé contient en tout 48352 CLB (= nombre d'Adaptive LUTs et registres)

2 cœurs: (temps de calcul: 171.92s - utilisations ALUTs: 8% - registres: 7%)



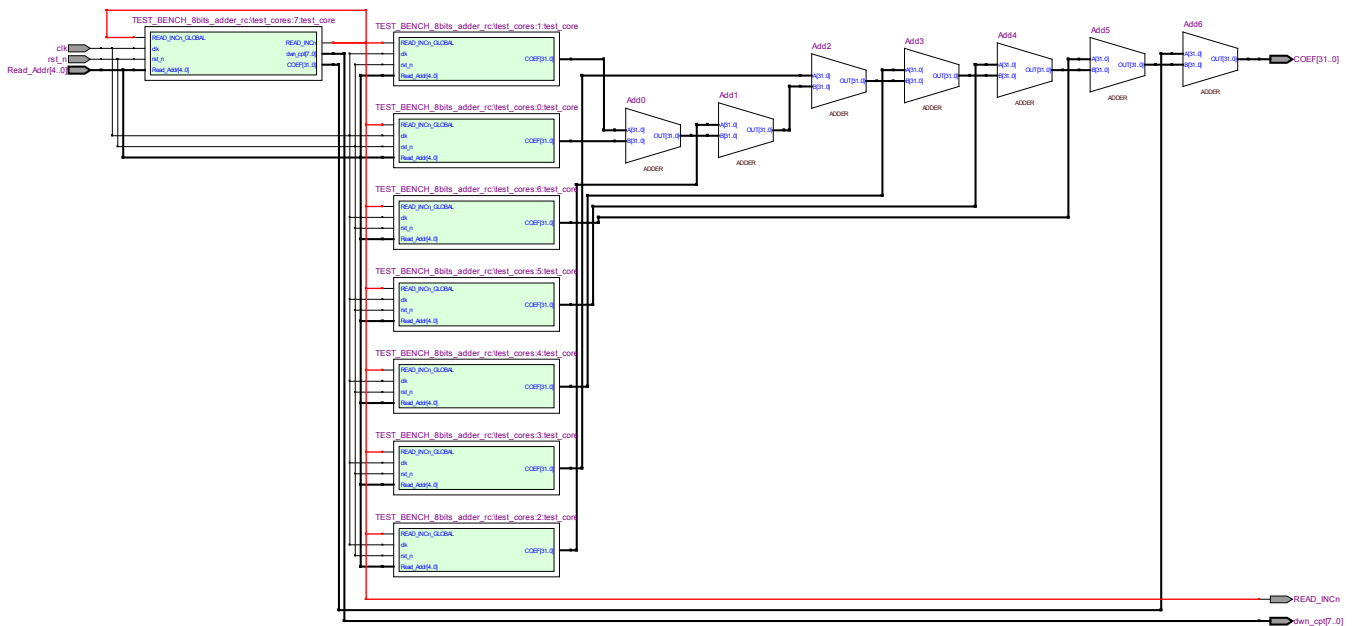
Compilation: 10 min

4 cœurs: (temps de calcul: 86.01s - utilisations ALUTs: 9% - registres: 10%)



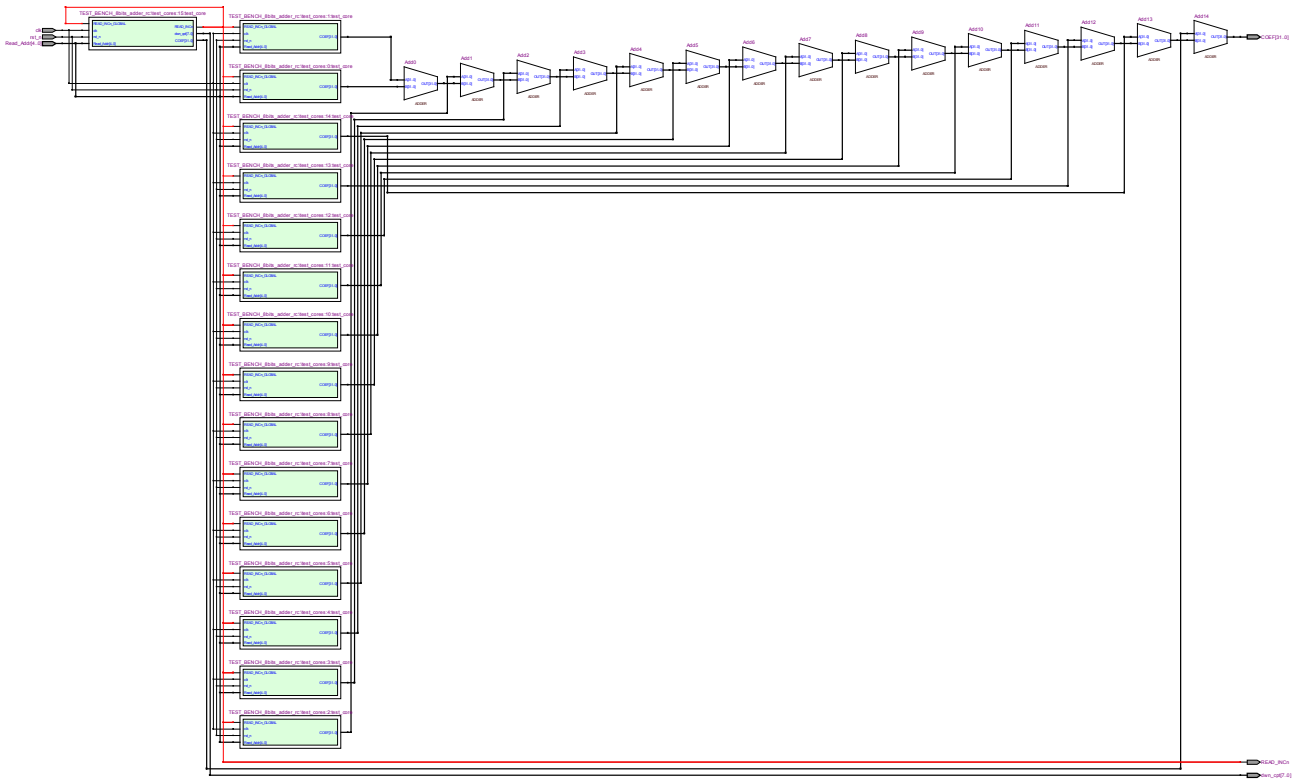
Compilation: 12 min

8 cœurs: (temps de calcul: 43.10s - utilisations ALUTs: 12% - registres: 15%)



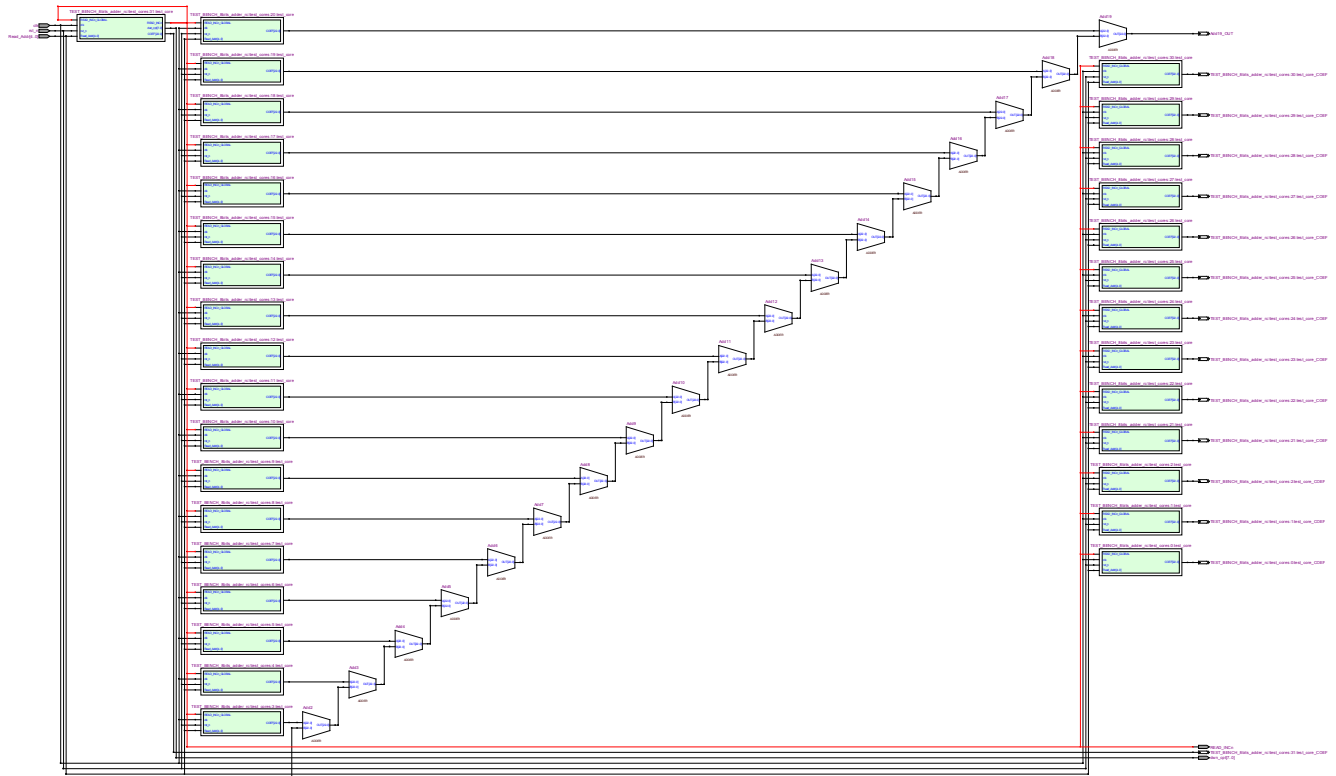
Compilation: 19min

16 cœurs: (temps de calcul: 21.64s - utilisations ALUTs: 15% - registres: 25%)



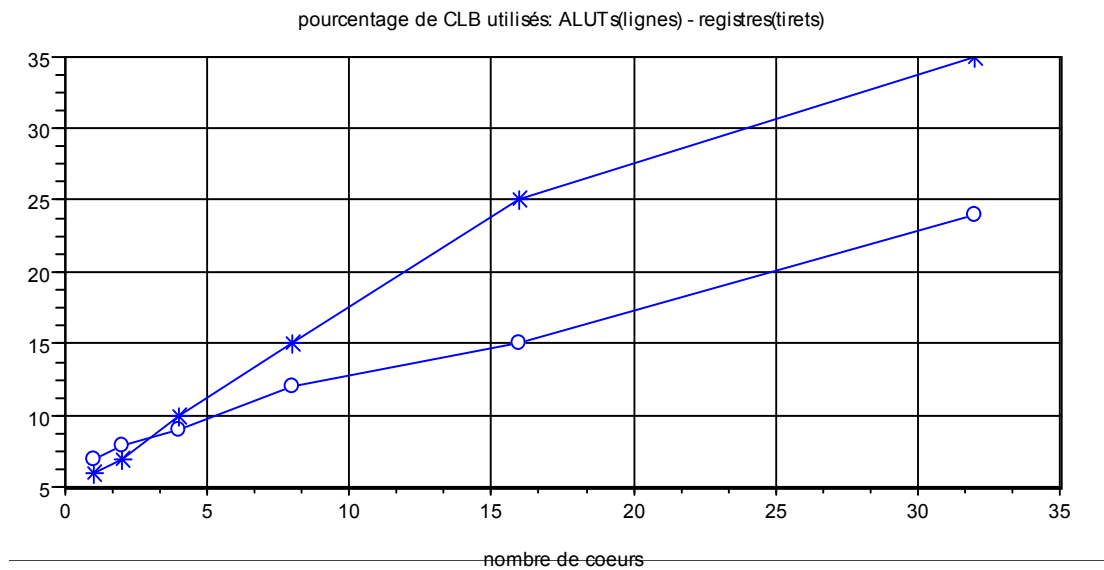
Compilation: 26min

32 cœurs: (temps de calcul: 10.87s - utilisations optimisée : ALUTs: 24% - registres: 35%)

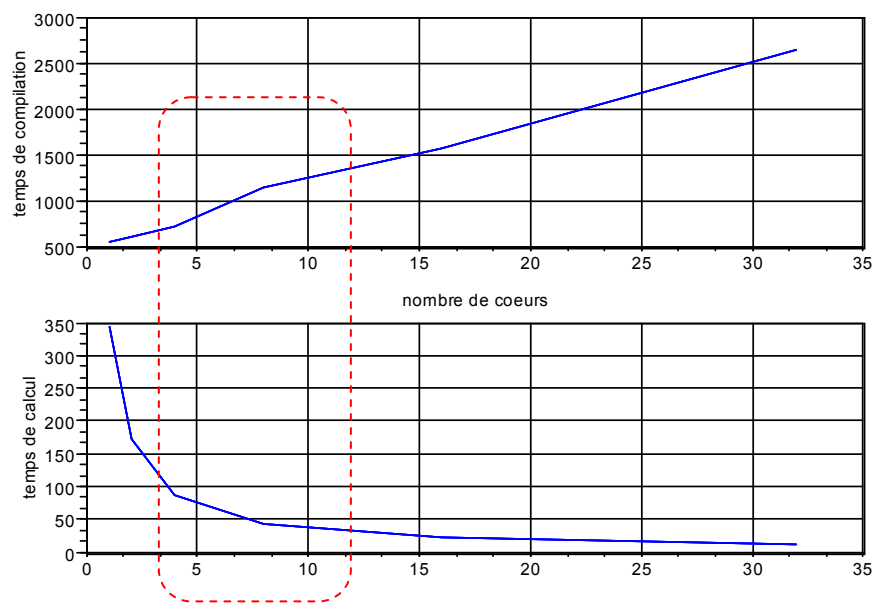


Compilation: 44min

Comparaison des test-bench « multi cœurs » : limites en surface (sur 48352 CLB)



Comparaison des test-bench « multi cœurs » : limites en temps de compilation / calcul



Ces exemples, pris dans le cas de caractérisation de l'additionneur à propagation de retenue 8bits, montrent qu'un "compromis de confort" est intéressant pour **4 à 8 cœurs**. La démultiplication en plus de 32 cœurs peut néanmoins être non négligeable, voir indispensable, pour les opérateurs massifs tels qu'un multiplieur 16 bits.

6) Références bibliographiques

- [1] Ketan N. Patel, Igor L. Markov and John P. Hayes, "Evaluating Circuit Reliability Under Probabilistic Gate-Level Fault Models" in *Proceedings of the International Workshop on Logic and Synthesis* (2003).
- [2] M. Correia de Vasconcelos, D. Teixeira Franco, L. Naviner and J.F. Naviner, "Reliability Analysis of Combinational Circuits Based on a Probabilistic Binomial Model" in IEEE-NEWCAS and TAISA Conference (2008).
- [3] A. Al-Yamani and E.J. McCluskey, "Seed Encoding with LFSRs and Cellular Automata," in *Proc. Design Automation Conference*, pp. 560-65 (2003).
- [4] E. Mollick. Establishing Moore's Law. *IEEE Annals of the History of Computing*, 28(3) : 62–75, Jul.-Sep. 2006.
- [5] P. K. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann Publishers, USA, (2001).
- [6] M. Nicolaidis. Design for Soft Error Mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3) : 405–418, Sep. 2005.
- [7] Michael Wirthlin, Nathan Rollins, Michael Caffrey, and Paul Graham. Hardness by design techniques for programmable gate arrays. *Proceedings of the 11th Annual NASA Symposium on VLSI Design*, 1, May 2003.
- [8] S. C. GOLDSTEIN and M. BUDIU. NanoFabrics : Spatial Computing Using Molecular Electronics. *Proceedings of the 28th International Symposium on Computer Architecture*, pages 178–189, Jun. 2001.
- [9] D. T. Franco, J.-F. Naviner, and L. Naviner. Yield and reliability issues in nanotechnologies. *Annales des télécommunications*, 61(11-12), Nov.-Dec. 2006.
- [10] Q. Zhou and K. Mohanram, "Gate sizing to radiation harden combinational logic," *IEEE Trans. Computer-aided Design*, vol. 25, pp. 155-166, Jan. 2006.
- [11] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra and J. Maiz, "Radiation Induced Soft Error Rates of Advanced CMOS Bulk Devices," *IEEE Intl. Reliability Physics Symp.*, 2006.