



EVALUATION PREDICTIVE DE LA SURETE DE FONCTIONNEMENT D'UN CIRCUIT INTEGRE NUMERIQUE

K. Hadjiat

► To cite this version:

K. Hadjiat. EVALUATION PREDICTIVE DE LA SURETE DE FONCTIONNEMENT D'UN CIRCUIT INTEGRE NUMERIQUE. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2005. Français. NNT : . tel-00009485

HAL Id: tel-00009485

<https://theses.hal.science/tel-00009485>

Submitted on 14 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
| / / / / / / / / / |

THESE

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro et Nano Electronique

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

Présentée et soutenue publiquement

Par

Karim HADJIAI

Le 10 juin 2005

Titre :

EVALUATION PREDICTIVE DE LA SURETE DE FONCTIONNEMENT D'UN CIRCUIT INTEGRE NUMERIQUE

Directeur de Thèse : Régis LEVEUGLE

JURY

M. Guy MAZARE
M. Matteo SONZA REORDA
M. Pascal FOUILLAT
M. Régis LEVEUGLE
M. Jean-Pierre SCHOELLKOPF

- , Président
- , Rapporteur
- , Rapporteur
- , Directeur de thèse
- , Examineur

Remerciements

Je souhaite m'adresser en premier lieu aux membres du jury de thèse. Je tiens à remercier Monsieur Matteo Sonza Reorda, Professeur au Politecnico de Turin, et Monsieur Pascal Fouillat, Professeur à l'université de Bordeaux, d'avoir accepté de rapporter sur mon travail, pour l'intérêt qu'ils ont porté à ce projet ainsi que pour leurs commentaires et suggestions.

J'adresse aussi mes remerciements à Monsieur Guy Mazaré, Professeur à l'ENSIMAG, pour m'avoir fait l'honneur de présider mon jury de thèse. Je remercie aussi Jean-Pierre Schoellkopf, chef du département ADT à STMicroelectronics Crolles, pour m'avoir fait l'honneur de faire partie de ce jury.

La présente thèse a été effectuée au laboratoire TIMA sur le site Viallet de l'Institut Polytechnique National de Grenoble (INPG). Je remercie Monsieur Bernard Courtois, directeur de ce laboratoire, pour m'y avoir accueilli.

Je tiens tout particulièrement à remercier mon directeur de thèse Monsieur Régis Leveugle, Professeur à l'ENSERG, pour m'avoir encadré tout au long de ces travaux sur un sujet porteur et passionnant. Je lui dois des voies de recherches originales, ainsi qu'une formation scientifique et méthodologique de qualité. Toute ma reconnaissance pour son soutien et sa patience qu'il a su m'accorder pour mener à bien ce manuscrit.

Toute ma gratitude au personnel fort sympathique de TIMA, CMP et du CIME : Professeurs, Maîtres de conférence, doctorants, administrateurs réseaux et secrétaires, pour leur aide et leur assistance.

Je souhaite maintenant adresser toute ma sympathie à mes amis et collègues du laboratoire TIMA. C'est à eux que je dois des moments inoubliables, passés dans un cadre amical, encourageant et enrichissant. Un grand merci donc à Abdelaziz, Abdelmajid, Achraf, Ahcene, Aimen, Amel, Diana, Ghiath, Iuliana, Lobna, Marius, Michele, Mohamed, Nacer, Pierre, Susi et Wassim.

Ma profonde reconnaissance à mes amis (et anciens collègues de TIMA), avec qui j'ai passé des années formidables, qui m'ont encouragé dans la bonne humeur : Amer, Amine, Faiza, Kamel, Khirddine, Menouer, Nadir et Samy.

Que celles et ceux que j'aurais oubliés ici me pardonnent.

Enfin, je ne remercierais jamais assez mes chers parents, mes frères ainsi que ma femme pour leur soutien tout au long de ce travail de longue haleine. Ma pensée va également à l'ensemble de ma famille, petits et grands, ainsi qu'à tous mes proches.

Table des matières

Introduction	9
Chapitre 1. Etat de l'art sur l'injection de fautes.....	11
1.1. Introduction à la notion de sûreté de fonctionnement d'un système	11
1.2. Catégories et modèles de fautes	15
1.2.1. Fautes de type matériel/physique.....	15
1.2.1.1. SEL	16
1.2.1.2. SEU et MBU	16
1.2.1.3. SET	17
1.2.1.4. Attaque par laser	18
1.2.1.5. Intégrité du signal	18
1.2.1.6. Modélisation des fautes transitoires.....	18
1.2.2. Fautes de type logiciel	21
1.3. Environnement d'injection de fautes	22
1.4. Injection de fautes basée sur le matériel.....	23
1.4.1. Injection de fautes niveau broches.....	24
1.4.2. Injection par particules.....	26
1.4.3. Injection de fautes par laser	27
1.4.4. Avantages et inconvénients.....	29
1.5. Injection de fautes basée sur le logiciel.....	30
1.5.1. Méthodes et outils	30
1.5.2. Avantages et inconvénients.....	35
1.6. Injection de fautes basée sur la simulation.....	36
1.6.1. Méthodes et outils	36
1.6.2. Avantages et inconvénients.....	40
1.7. Prototypage matériel	41
1.7.1. Méthodes.....	41
1.7.2. Avantages et inconvénients.....	43
1.8. Injection de fautes hybride	44
1.9. Comparaison entre les techniques d'injection de fautes	46
1.10. Conclusion	47

Chapitre 2. Flot d'analyse proposé	49
2.1. Objectifs du travail.....	49
2.2. Flot global.....	49
2.2.1. Vue d'ensemble	50
2.2.2. Simulation versus émulation.....	52
2.2.3. Saboteurs versus mutants.....	54
2.3. Résultats d'analyse	55
2.3.1. Classification.....	55
2.3.2. Génération d'un modèle comportemental de propagation d'erreur.....	55
2.3.3. Une approche hiérarchique	58
2.3.4. Portabilité et raffinement bas niveau du modèle généré.....	58
2.4. Conclusion	59
Chapitre 3. Injection de fautes simples "SEU"	61
3.1. Limitation des approches précédentes de génération de mutants.....	61
3.2. Génération contrôlée de mutants pour l'injection de transitions erronées.....	62
3.2.1. Différentes alternatives pour modifier une description VHDL	63
3.2.1.1. DFS-SE	64
3.2.1.2. DFS-SN.....	66
3.2.1.3. IFS-SE et IFS-SN.....	67
3.2.2. Généralisation de l'étude sur des registres d'état de type entier.....	68
3.2.3. Evaluation qualitative des surcoûts de mutation sur des exemples de circuit	69
3.2.4. Utilisation d'informations partielles de synthèse.....	70
3.3. Génération de mutants pour les inversions de bits.....	71
3.4. Contrôle externe des instants d'injection	72
3.5. Extension aux mémoires RAM et bancs de registres	74
3.6. Modifications structurelles induites par les mutants	76
3.6.1. Transitions erronées	76
3.6.2. Inversion de bits pour les registres de donnée ou de contrôle	77
3.6.3. RAM ou banc de registres.....	78
3.6.4. Le cas des registres non ciblés par l'injection	80
3.7. Conclusion	82
Chapitre 4. Injection de fautes multiples "MBF"	83
4.1. Génération hétérogène des mutants.....	83
4.2. Les différentes alternatives pour modifier une description VHDL	85
4.2.1. Architecture globale.....	85
4.2.2. Injections multiples limitées à un seul registre.....	86
4.2.3. Injections multiples sans limitation spatiale à l'intérieur d'un sous-bloc.....	89
4.2.4. Modifications structurelles induites par les mutants.....	94
4.3. Gestion hiérarchique	95
4.4. Compromis entre la généralité et la complexité.....	101

4.5. Conclusion	102
Chapitre 5. Expériences d'injection.....	103
5.1. Injections multi-niveaux.....	103
5.1.1. Co-processeur de multiplication entière	103
5.1.1.1. Descriptions	103
5.1.1.2. Injection de SEU	105
5.1.2. Processeur Fuzzy	109
5.1.2.1. Description.....	109
5.1.2.2. Injection de SEU	111
5.2. Injection de fautes multiples "MBF"	112
5.2.1. Co-processeur de multiplication entière	112
5.2.1.1. Surcoûts	112
5.2.1.2. Expériences d'injection.....	115
5.2.2. Co-processeur de multiplication modulaire	116
5.2.2.1. Description.....	116
5.2.2.2. Expériences d'injection.....	117
5.3. Conclusion sur les expériences d'injection	119
Conclusion et perspectives	121
Annexe A	125
Annexe B	127
Références bibliographiques	129
Publications.....	138

Liste des Figures

Figure 1-1: Taxonomie de la sûreté de fonctionnement.....	13
Figure 1-2: Les attributs perceptifs de la sûreté de fonctionnement.....	14
Figure 1-3: Scénario de propagation d’une impulsion transitoire dans un circuit combinatoire (a) et les formes d’onde des entrées de bascules (b).....	17
Figure 1-4: Génération de fautes pendant le procédé de développement d’un logiciel.....	21
Figure 1-5: Les composants de base d’un environnement d’injection de fautes.....	23
Figure 2-1: Vue globale du flot d’analyse.....	50
Figure 2-2: Flot global d’analyse pour une description modifiée de circuit.....	51
Figure 2-3: Exemple de modèle obtenu en classifiant l’effet des fautes	55
Figure 2-4: Exemple de modèle obtenu en analysant les chemins de propagation des erreurs.....	56
Figure 3-1: Description VHDL d’un FSM simple	63
Figure 3-2: Insertion des trois signaux d’entrées dans l’entité FSM définie en figure 3-1	65
Figure 3-3 : Fonction "std_to_state" générée pour l'exemple FSM.....	65
Figure 3-4: Fonction "affect" générée pour l'exemple FSM.....	66
Figure 3-5: Insertion des trois signaux d’entrée dans l’entité FSM.....	66
Figure 3-6: Injection dans l’approche DFS-SN pour l’exemple FSM.....	67
Figure 3-7: Utilisation explicite du modulo pour le mode IFS-SE.....	67
Figure 3-8: Utilisation explicite du modulo pour le mode IFS-SN	68
Figure 3-9: Fonction de conversion "std_to_state"	69
Figure 3-10: Application du modèle des inversions de bits uniques sur le registre "state".....	71
Figure 3-11: Exemple de conversion de type.....	72
Figure 3-12: Protocole d’injection (injection active sur front montant ou descendant).....	72
Figure 3-13: Injection au milieu de la 2 ^{ème} moitié de la période	73
Figure 3-14: Génération du signal Clk_reg.....	73
Figure 3-15: La partie séquentielle d’un FSM muté (injection asynchrone pour le modèle des inversions de bits uniques).....	74
Figure 3-16: Description d'une RAM qui regroupe 4 mots de 2 bits chacun	75
Figure 3-17: Modification de la RAM pour des inversions de bit uniques	75
Figure 3-18: Machine d’états avant modification.....	76
Figure 3-19: Machine d’états après modification.....	77
Figure 3-20: Inversion de bits dans un registre cible "Ri".....	78
Figure 3-21: Représentation structurelle du surcoût induit pas l’instrumentation du banc de registres....	79
Figure 3-22: Ajout d’une entrée pour assurer la mémorisation.....	80

Figure 3-23: Combinaison entre le signal d'injection et l'entrée de validation	80
Figure 3-24: Retranscription des instructions d'affectation des registres non ciblés par l'injection.....	81
Figure 3-25: Insertion des entrées de validation pour les registres non ciblés par l'injection.....	81
Figure 4-1: Exemples d'attaques par laser selon (a) un placement/routage 1 et une focalisation 1 (b) un placement/routage 2 et une focalisation 1 (c) le placement/routage 2 et une focalisation 2.	84
Figure 4-2: Architecture globale d'un mutant pour modèle de fautes hétérogène	86
Figure 4-3 : Injection localisée de MBF dans les registres (a) R1 et (b) R3, respectivement de multiplicité deux et quatre	86
Figure 4-4: Insertion des signaux de contrôle dans l'entité.....	87
Figure 4-5: Modification du processus séquentiel.....	88
Figure 4-6: Insertion du processus combinatoire qui réalise l'inversion multiple.	89
Figure 4-7 : Injection généralisé de MBF par le biais du registre virtuel.....	90
Figure 4-8: Insertion des signaux de contrôle dans l'entité.....	90
Figure 4-9: Modification du processus séquentiel.....	91
Figure 4-10: Modification du processus séquentiel dans le composant d'une description hiérarchique. ...	91
Figure 4-11: Insertion du processus qui réalise l'inversion multiple.	92
Figure 4-12: Insertion du processus qui réalise l'inversion multiple.	93
Figure 4-13: Représentation structurelle des modifications pour des injections de type MBF	94
Figure 4-14: Description hiérarchique modifiée pour l'injection de fautes de type MBF.	96
Figure 4-15: Insertion des signaux de contrôle dans les différents blocs de la hiérarchie	97
Figure 4-16: Fonctions utilisées pour désactiver les composants non concernés par l'injection asynchrone. Dans (a) la profondeur hiérarchique est égale à 1 alors que dans (b) elle est supérieure à 1.	98
Figure 4-17: Protocole d'injection.....	99
Figure 4-18: Représentation structurelle d'une description hiérarchique de profondeur 2 modifiée pour l'injection de fautes type MBF.....	100
Figure 4-19: Fonctions utilisées pour gérer l'injection asynchrone. Les processus décrits dans (a) et (b) seront respectivement insérés dans les descriptions des composants B0 et B1.....	101
Figure 5-1: Spécification de l'état "multin" de la description "architecture".....	103
Figure 5-2: Structure hiérarchique du multiplieur.....	104
Figure 5-3: Une partie de la description "micro-architecture" avec (a) la partie contrôle.....	104
Figure 5-4: Evolution de la surface en fonction de la multiplicité (implémentation ASIC)	114
Figure 5-5: Evolution du chemin critique en fonction de la multiplicité (implémentation ASIC).....	114
Figure 5-6: Evolution du nombre de cellules "slices" en fonction de la multiplicité (implémentation FPGA)	114
Figure 5-7: Evolution du chemin critique en fonction de la multiplicité (implémentation FPGA).....	114
Figure 5-8: Schéma bloc du multiplieur de Montgomery	117

Liste des Tableaux

Tableau 1-1: Résumé des avantages et inconvénients des différentes techniques d'injection de fautes...	46
Tableau 3-1: Résumé des résultats obtenus à partir des benchmarks ITC99 pour la génération de mutants (surcoût en complexité, diminution de la fréquence d'horloge et augmentation du nombre d'E/S par rapport à la description initiale) [R.L. 00]	70
Tableau 3-2: Surtout en complexité, fréquence d'horloge et augmentation du nombre d'E/S par rapport à la description initiale d'un banc de registres	80
Tableau 5-1: Résultat de Synthèse pour la spécification "architecture" et "micro-architecture" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).....	105
Tableau 5-2: Caractéristiques des modèles comportementaux générés (observation des signaux externes seulement, aucun état terminal).....	106
Tableau 5-3: Caractéristiques des modèles comportementaux générés (observation des signaux externes et internes, aucun état terminal).....	106
Tableau 5-4: Caractéristiques des modèles comportementaux générés avec spécification des états terminaux.....	108
Tableau 5-5: Comparaison des résultats d'analyse entre le plus haut et le plus bas niveau - nombre de transitions communes des modèles comportementaux obtenus après injection des transitions erronées dans la description "architecture" et des inversions de bit dans la description "micro-architecture"	108
Tableau 5-6: Résultat de Synthèse pour la spécification "architecture 2" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).....	108
Tableau 5-7: Caractéristiques des modèles comportementaux générés (observation des signaux externes seulement, aucun état terminal).....	109
Tableau 5-8: Caractéristiques des modèles comportementaux générés (observation des signaux externes et internes, aucun état terminal).....	109
Tableau 5-9: Les différents processus inclus dans la description "micro-contrôleur".	110
Tableau 5-10: Résultat de Synthèse pour la spécification "contrôleur" et "micro-contrôleur" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).....	110
Tableau 5-11: Résultats d'injection de SEU sur le registre "counter".....	111
Tableau 5-12: Résultats d'injection de SEU sur le registre d'états "state".....	111
Tableau 5-13: Résultats obtenus après génération des graphes comportementaux.....	112
Tableau 5-14: Evolution de la complexité avec la multiplicité (pour une implémentation ASIC).	113
Tableau 5-15: Evolution de la complexité avec la multiplicité (en ressources FPGA).	113
Tableau 5-16: Résultat d'analyse de la propagation d'erreur pour les différentes valeurs de multiplicité.	116
Tableau 5-17: Définition par bloc des cibles d'injection	117
Tableau 5-18: Pourcentages des cas conduisant à l'état "crash" pour différentes valeurs de la multiplicité	119
Tableau 5-19: Résultats de l'analyse de propagation d'erreurs pour différentes valeurs de la multiplicité	119

Introduction

L'évolution des technologies CMOS vers des dimensions inférieures à 0,12 micron, conjuguée avec la forte diminution des tensions d'alimentation, rend les circuits intégrés de plus en plus sensibles à des phénomènes pouvant induire des erreurs de fonctionnement. En particulier, un taux non négligeable d'erreurs est attendu d'ici quelques années, même pour les applications terrestres, à cause des fautes de type "SEU" (inversion de bits occasionnée dans les points mémoires internes d'un circuit par des rayonnements ou des impacts de particules). Par exemple, il est intéressant de noter que les neutrons atmosphériques, jusqu'à présent inoffensifs pour les circuits micro-électroniques, ont des énergies suffisantes pour faire basculer le contenu de cellules mémoires ou même perturber les opérations logiques dans des circuits fabriqués dans des technologies submicroniques. Ceci peut constituer une source sérieuse d'erreurs, particulièrement dans l'avionique (à 30 000 pieds le flux de neutrons est de 4 à 8 fois plus important qu'au sol) mais aussi dans les systèmes opérant au sol.

L'augmentation de la probabilité de telles fautes transitoires peut rendre nécessaire leur détection ou leur tolérance dans un nombre croissant de circuits intégrés, y compris pour des applications n'ayant pas traditionnellement mis en oeuvre ce type de technique. Pour maintenir la fiabilité minimale exigée par les utilisateurs, on peut s'attendre, par exemple, dans les années qui viennent à la mise en oeuvre de techniques de tolérance aux fautes dans tous les domaines de l'électronique mobile : ordinateurs portables, téléphones portables, agenda électronique, etc... De telles applications étant cependant soumises à de nombreuses contraintes de conception, des compromis seront à établir entre l'augmentation de fiabilité et la dégradation des autres paramètres (coût, consommation, fréquence de fonctionnement, ...).

L'intégration de mécanismes de détection ou de tolérance nécessitera, en premier lieu, d'analyser l'impact réel que peut avoir une faute donnée sur le comportement global du circuit, afin de déterminer sa criticité et de décider de l'ajout éventuel d'un dispositif de protection. L'analyse des fonctionnements aberrants pouvant résulter d'un SEU est toutefois un processus complexe et coûteux en temps. Par ailleurs, il est important de pouvoir réaliser cette analyse le plus tôt possible dans le processus de conception. Pour cela, il est possible d'utiliser une approche fondée sur l'injection de fautes dans des descriptions de haut niveau (classiquement, une description VHDL au niveau RTL) et la génération d'un modèle représentant les comportements consécutifs à la faute. Ce modèle comportemental est dérivé de résultats de simulations ou d'exécutions sur un prototype matériel. Il doit mettre en évidence les chemins de propagation des erreurs dans le circuit et fournir des indications quantitatives sur les probabilités d'occurrence des différents modes de défaillance. Une telle analyse est indispensable pour décider de la stratégie de tolérance la mieux adaptée pour chaque circuit (type de mécanisme à

implémenter et sites à protéger), et sa complexité impose de disposer d'outils automatisant l'essentiel du travail.

Dans ce contexte, deux aspects complémentaires sont considérés dans le cadre de la thèse : l'injection de fautes et l'analyse des résultats obtenus à l'issue de la campagne d'injection.

Le chapitre 1 du manuscrit présente l'état de l'art sur l'injection de fautes. Nous commencerons par introduire des notions de base concernant la sûreté de fonctionnement d'un système numérique. Ensuite les techniques d'injection de fautes sont détaillées. Différentes approches se basent sur le matériel, sur le logiciel ou sur de la simulation mais visent toutes à évaluer la sûreté de fonctionnement de circuits intégrés. Pour chacune de ces techniques d'injection, nous présentons les limitations, les avantages et les inconvénients, ainsi que les travaux existant et les différents outils développés pour l'étude du comportement des circuits numériques en présence d'erreurs. Pour conclure ce premier chapitre, nous comparons succinctement les différentes techniques d'injection.

Le chapitre 2 présente, à travers un flot global, la méthodologie adoptée pour effectuer les injections de fautes et l'analyse des résultats obtenus. En fonction des zones étudiées dans un modèle HDL de haut niveau, un modèle comportemental est généré automatiquement pour représenter le comportement du circuit consécutif à l'injection de fautes.

Dans le chapitre 3, l'accent est mis sur la génération automatisée de spécifications VHDL mutées. Les mutants générés permettent d'effectuer des injections de fautes de type SEU dans tous les points mémoire du circuit, tout en étant synthétisables de façon efficace. Ils permettent de plus de prendre en compte plusieurs niveaux de modélisation des fautes.

Les campagnes d'injection classiques se basent sur le modèle des inversions de bits uniques. Nous discutons dans la section 1.2.1.6 l'intérêt d'étendre ce modèle aux inversions multiples. La génération de mutants VHDL pour ce modèle étendu est présentée dans le chapitre 4, et nous montrons comment plusieurs modèles de fautes peuvent être combinés et inclus dans une seule campagne d'injection par le biais d'une génération hétérogène de mutant. Dans le cas d'expériences basées sur l'émulation, des compromis entre la complexité et la généralité sont également étudiés.

Dans le chapitre 5, les différentes approches proposées seront validées sur des exemples d'application. La première partie sera consacrée aux injections de fautes simples dans un coprocesseur arithmétique et un processeur Fuzzy. Nous considérons des descriptions de circuits et des modèles de fautes multi-niveaux. Les résultats démontrent l'intérêt d'une évaluation de sûreté réalisée très tôt dans le processus de conception. La seconde partie porte sur l'injection de fautes multiples sur le coprocesseur arithmétique et un coprocesseur de multiplication modulaire. Nous montrons l'influence de la multiplicité spatiale sur le comportement du circuit cible.

Chapitre 1.

Etat de l'art sur l'injection de fautes

Nous introduisons dans ce chapitre les notions de base sur la sûreté de fonctionnement d'un système. Ensuite, nous présentons les principales techniques d'injection de fautes appliquées sur différents types de circuits. Les avantages et limitations des différentes techniques d'injection sont présentés et comparés.

1.1. Introduction à la notion de sûreté de fonctionnement d'un système

Les notions présentées ici sont extraites de [A.A. 04-a], [J.L. 88]. Elles permettront au lecteur de bien identifier la terminologie employée dans cette thèse.

La **sûreté de fonctionnement** d'un système Logiciel/Matériel est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il délivre. Le service délivré par un système est son comportement tel que perçu par son, ou ses, utilisateurs ; un utilisateur du système est un autre système (humain ou physique) qui interagit avec celui-ci.

Une **défaillance** du système survient lorsque le service délivré dévie des conditions établies dans la **spécification**, qui est une description agréée du service attendu. Une **erreur** est un état, par rapport au processus de traitement, susceptible de conduire à une défaillance. Une **faute** est la cause supposée ou adjugée d'une erreur. Une erreur est donc la manifestation d'une faute dans le système, alors qu'une défaillance est la manifestation d'une erreur sur le service délivré par le système.

Une erreur peut résulter :

- de l'activation d'une faute interne au système, préalablement dormante ; une faute interne peut :
 - résulter d'une défaillance physique d'un composant matériel (alors vu comme un système) ; on parle alors de faute physique.
 - être une faute de conception affectant le logiciel ou le matériel ; le terme "conception" doit être pris ici au sens large, depuis l'expression des besoins du système jusqu'à sa réalisation, tant au cours de la production initiale du système que lors de modifications ultérieures.

- de l'occurrence d'une faute externe, qui peut soit être relative à l'environnement physique du système, soit être une faute humaine d'interaction ; il y a alors génération d'une erreur.

L'obtention d'un système Logiciel/Matériel sûr de fonctionnement passe par l'utilisation combinée d'un ensemble de méthodes que l'on peut classer comme suit :

- **Prévention des fautes** : comment empêcher, par construction, l'occurrence ou l'introduction de fautes.
- **Tolérance aux fautes** : comment fournir, par redondance, un service conforme à la spécification en dépit des fautes.
- **Élimination des fautes** : comment minimiser, par vérification, la présence de fautes.
- **Prévision des fautes** : comment estimer, par évaluation, la présence, la création, et les conséquences de fautes.
- Prévention des fautes et élimination des fautes peuvent être vues comme constituant l'**évitement des fautes** : comment tendre vers un système exempt de fautes.
- Prévention des fautes et tolérance aux fautes peuvent être vues comme constituant l'**obtention de la sûreté de fonctionnement** : comment procurer au système l'aptitude à délivrer un service conforme à la spécification.
- Élimination des fautes et prévision des fautes peuvent être regroupées dans la **validation de la sûreté de fonctionnement** : comment avoir confiance dans l'aptitude du système à délivrer un service conforme au service spécifié.

La vie d'un système est perçue par ses utilisateurs comme une alternance entre deux états du service délivré par rapport au service spécifié : service approprié et service inapproprié, la transition du service approprié au service inapproprié étant une défaillance. La quantification de cette alternance conduit aux deux mesures principales de la sûreté de fonctionnement :

- **La fiabilité** : mesure de la délivrance continue d'un service approprié, ou, ce qui est équivalent, mesure du temps jusqu'à défaillance.
- **La disponibilité** : mesure de la délivrance d'un service approprié par rapport à l'alternance service approprié - service inapproprié.

Un système ne défaille généralement pas toujours de la même manière, ce qui conduit à distinguer les modes et sévérités de défaillance.

Deux larges classes de modes de défaillance sont :

- les défaillances contrôlées : le mode de défaillance appartient à un sous-ensemble strict des modes de défaillance possibles, sous-ensemble qui est spécifié.
- les défaillances incontrôlées : le mode de défaillance est quelconque.

Les sévérités des défaillances résultent du classement des conséquences des modes de défaillance sur l'environnement du système. Elles permettent donc d'ordonner les modes de défaillance. Un cas particulier, souvent rencontré dans la pratique, est celui de systèmes dont les modes de défaillance peuvent être groupés en deux classes dont les sévérités sont considérablement différentes : défaillances bénignes et défaillances malignes, ou catastrophiques. Un système dont toutes les défaillances sont, éventuellement dans une mesure acceptable, des défaillances bénignes, est dit système à défaillance saine (en anglais, "fail-safe"). Une mesure du temps jusqu'à défaillance catastrophique est la **sécurité**.

Les notions introduites jusqu'à présent constituent des attributs de la sûreté de fonctionnement, qui peuvent être groupés en trois classes

- les **entraves** à la sûreté de fonctionnement, qui sont des circonstances indésirables, mais non imprévues, cause ou résultat de la non-sûreté de fonctionnement (dont la définition se déduit simplement de celle de la sûreté de fonctionnement : une confiance justifiée ne peut plus, ou ne pourra plus, être placée dans le service délivré)
- les **moyens** pour la sûreté de fonctionnement, qui sont les méthodes, outils et solutions permettant de procurer au système l'aptitude à délivrer un service de confiance, et d'avoir confiance dans cette aptitude.
- les **mesures** de la sûreté de fonctionnement, qui permettent d'apprécier la qualité du service délivré, telle que résultant des entraves et des moyens de s'y opposer.

Ces attributs sont résumés sous forme d'un arbre taxonomique par la Figure 1-1.

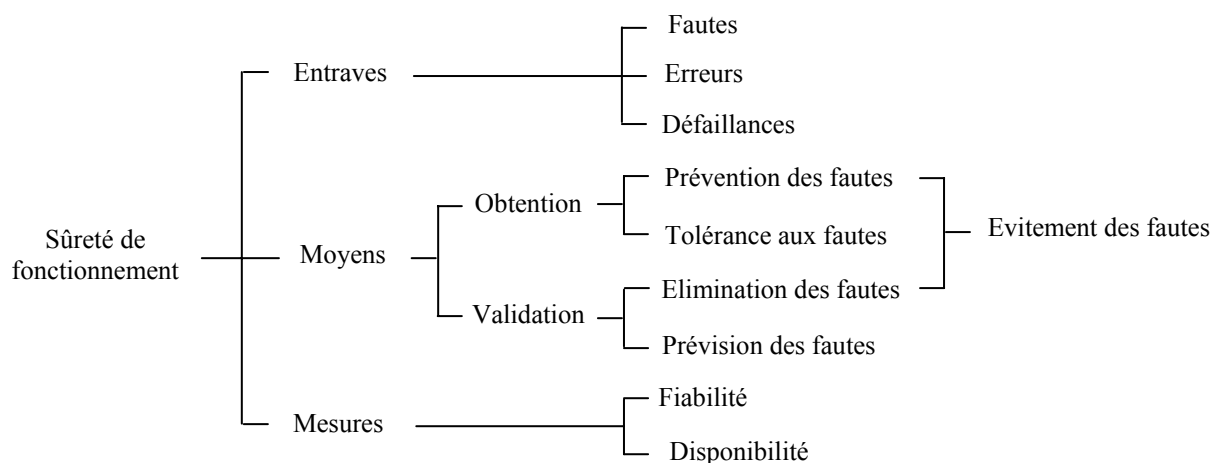


Figure 1-1: Taxonomie de la sûreté de fonctionnement.

Une vision plus complète de la sûreté de fonctionnement est obtenue en la considérant selon différents points de vue, permettant de définir ses attributs perceptifs :

- par rapport à *la continuité du service*, la sûreté de fonctionnement est perçue comme la **fiabilité**.
- par rapport à *la non-occurrence de défaillances catastrophiques*, la sûreté de fonctionnement est perçue comme la sécurité ("safety" en anglais).
- par rapport à *la préservation de la confidentialité et de l'intégrité* des informations, la sûreté de fonctionnement est perçue comme la sécurité ("security" en anglais).

Ces attributs perceptifs, qui sont illustrés par la Figure 1-2, peuvent figurer en tout ou partie dans la spécification, selon les buts et la nature de système considéré.

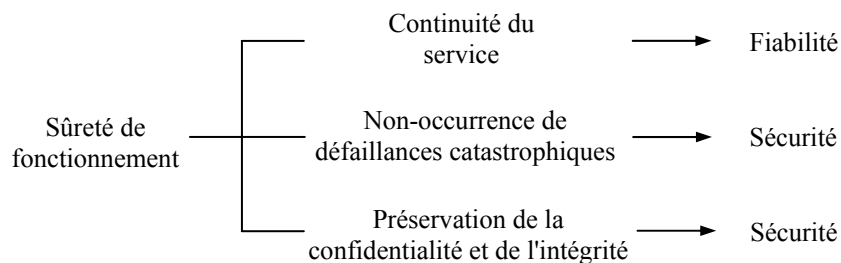


Figure 1-2: Les attributs perceptifs de la sûreté de fonctionnement.

En fait, les relations entre fiabilité, sécurité au sens des défaillances catastrophiques et sécurité au sens de la confidentialité sont plus complexes qu'une simple dépendance. Le premier souci fut de faire fonctionner les systèmes informatiques (fiabilité). Leur utilisation dans des applications critiques amène le souci de sécurité. Un système qui ne fait rien ne peut avoir des défaillances catastrophiques, mais n'est pas très utile. D'où la prétention de certains cercles de la sécurité au sens des défaillances catastrophiques de considérer la fiabilité comme un sous-ensemble de leur spécialité. L'avènement des systèmes informatiques répartis a attiré l'attention sur les fautes intentionnelles ; ces dernières peuvent avoir des conséquences catastrophiques, d'où la répétition du phénomène précédemment mentionné (les spécialistes de la sécurité au sens de la confidentialité considèrent généralement fiabilité et sécurité au sens des défaillances catastrophiques comme des sous-ensembles de leur spécialité).

Il n'est pas important de savoir si la fiabilité est un sous-ensemble de la sécurité au sens des défaillances catastrophiques ou vice-versa, et il en est de même lorsque l'on considère en plus la sécurité au sens de la confidentialité. Ce qui importe réellement dans les relations entre fiabilité, sécurité au sens des défaillances catastrophiques et sécurité au sens de la confidentialité, est qu'elles caractérisent selon des points de vue différents la même propriété d'un système informatique : la confiance que l'on peut placer dans le service qu'il délivre, ce que nous appelons la sûreté de fonctionnement.

1.2. Catégories et modèles de fautes

Une faute est considérée comme une déviation d'une fonction prévue au niveau matériel ou logiciel. Elle peut surgir à n'importe quelle étape du processus de conception d'un système : spécification, conception, développement, fabrication, assemblage et installation du système dans son environnement opérationnel. Par ailleurs une faute peut éventuellement survenir durant la vie opérationnelle d'un système.

La plupart des fautes qui se produisent avant le déploiement complet du système doivent être découvertes par le test et éliminées. Celles qui ne sont pas enlevées peuvent réduire la fiabilité du système lorsqu'il est incorporé dans son environnement opérationnel [J.C. 95].

Les fautes systématiques sont liées d'une part à la phase de conception (oubli d'un composant, d'une connexion... dans le matériel ; erreur dans une instruction logicielle ...) et d'autre part à la fabrication (mauvaise gravure dans un circuit intégré, piste coupée sur un circuit imprimé...). Elles sont dues à des défaillances soit humaines (défaillance du concepteur), soit physiques (défaillance des machines de fabrication, mauvaise qualité de produits servant à la fabrication...).

Les fautes aléatoires apparaissent en phase d'exploitation. Ce sont les fautes internes, intrinsèques au système et dues à une défaillance d'un composant, par exemple d'une interconnexion, ou encore dues à des phénomènes externes tels que des parasites électromagnétiques ou des particules atmosphériques.

1.2.1. Fautes de type matériel/physique

Les fautes de type matériel qui surgissent pendant l'exploitation du système sont classifiées par leur durée : permanente, transitoire, ou intermittente.

Une faute permanente est causée par un dommage irréversible au niveau du composant, tel qu'une jonction de semi-conducteur qui a été court-circuitée en raison d'un vieillissement thermique. Une faute permanente dans un circuit entraîne en général la défaillance de la carte utilisant ce circuit, et le rétablissement du service peut seulement être obtenu en remplaçant ou en réparant le composant ou le sous-ensemble endommagé.

Une faute transitoire (ou temporaire) est déclenchée par des conditions environnementales telles que la fluctuation d'une ligne de courant, une interférence électromagnétique ou un impact de particule. Ces fautes sont généralement de courte durée et endommagent rarement le composant affecté, bien qu'elles puissent induire un état incorrect dans le système. Avec l'arrivée des technologies nanométriques, les fautes transitoires, déjà difficiles à modéliser et à prédire, sont aujourd'hui de plus en plus évidentes et nombreuses, et constituent la cause de plus de 80% des défaillances dans les systèmes informatiques [D.S. 92]. Parmi les effets transitoires les plus difficiles à maîtriser et dont l'impact est le plus difficile à évaluer, on note l'effet des particules provenant du rayonnement cosmique (par exemple, les neutrons atmosphériques) ou

de la désintégration des isotopes instables présents dans les matériaux des circuits intégrés et de leur encapsulation (particules alpha).

Les fautes temporaires internes sont souvent dénommées **fautes intermittentes**. Ces fautes résultent de la présence de combinaisons survenant rarement, telles que des changements au niveau des paramètres d'un composant matériel (effet de variations de température, retard dans les temporisations du à des capacités parasites) ou des situations, affectant aussi bien le logiciel que le matériel, survenant quand la charge excède une certaine valeur, par exemple des temporisations ou des synchronisations marginales.

Dans ce qui va suivre, nous allons nous focaliser sur les fautes transitoires provoquées par le passage d'une particule chargée à travers des zones sensibles du circuit. Plus les particules ont une masse et une énergie importante, plus elles provoquent de dommages. Les particules chargées (ions et particules alpha) sont les plus dangereuses ; elles sont majoritairement responsables des **SEEs** (Single Event Effects). On peut distinguer trois types de SEE présentés dans les sections suivantes. Par ailleurs, d'autres sources de fautes peuvent produire des effets similaires ; elles seront brièvement présentées, pour conduire à une modélisation de fautes transitoires aussi générale que possible.

1.2.1.1. SEL

Les SELs (Single Event Latchup) résultent de la mise en conduction de thyristors parasites présents dans les circuits CMOS et provoquent des courts-circuits susceptibles d'endommager le composant s'il n'est pas mis hors tension rapidement [A.D. 81].

1.2.1.2. SEU et MBU

On appelle **SEU** (Single Event Upset) un effet d'ionisation dans le matériau résultant du passage d'une particule chargée dans un circuit, phénomène qui peut causer un "upset" (aussi appelé "bit-flip" dans la littérature anglaise). L'upset est en fait le changement de l'information contenue dans un point mémoire en son inverse [E.P. 83]. La particule incidente produit des paires électrons-trous le long de sa trajectoire. L'impulsion de courant produite dans un nœud change son potentiel. Si cette impulsion produit une variation de tension suffisante pour provoquer la modification de l'état du transistor, le nœud changera d'état. Si cet état est maintenu suffisamment longtemps pour que l'effet de la mémorisation intervienne, il y aura un basculement logique [J.P. 83]. L'upset est un phénomène non destructif et réversible (par exemple, le point mémoire pourra être corrigé par le processus normal d'écriture).

Tout composant électronique possédant des points de mémorisation est sensible au phénomène d'upset (mémoires dynamiques ou statiques, microprocesseurs et périphériques de microprocesseurs, bascules, verrous...). Traditionnellement, dans le domaine spatial, les ions lourds étaient principalement considérés comme la cause des upsets. Les technologies à forte densité d'intégration et la diminution de la taille des transistors ont entraîné un accroissement de la sensibilité aux upsets de telle sorte que les protons et les neutrons atmosphériques sont

susceptibles d'en produire dans les circuits fonctionnant en environnement avionique, voire au niveau de la mer [C.C. 02], [P.S. 02], [E.N. 96].

Les **MBUs** (Multiple-Bit Upset) apparaissent du fait des charges qui diffusent dans le substrat, pouvant ainsi être collectées par plusieurs éléments du circuit. C'est l'exemple des mémoires DRAMs et SRAMs où plusieurs cellules peuvent changer d'état logique simultanément.

1.2.1.3. SET

Les **SETs** (Single Event Transient) résultent d'une impulsion transitoire créée par le passage d'une particule chargée dans la logique combinatoire d'un circuit numérique. Un cas typique de circuit logique affecté par une impulsion transitoire est présenté dans la Figure 1-3.

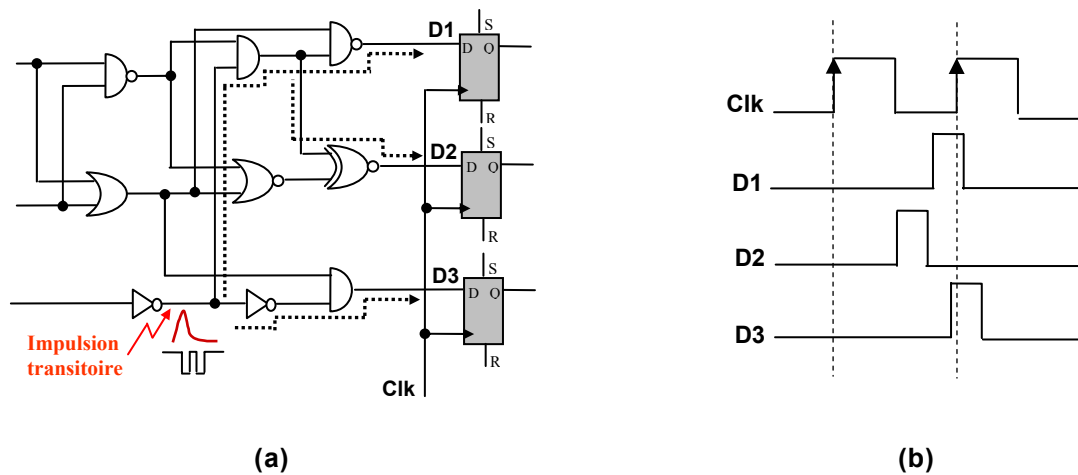


Figure 1-3: Scénario de propagation d'une impulsion transitoire dans un circuit combinatoire (a) et les formes d'onde des entrées de bascules (b).

Les trois chemins de propagation montrés dans la Figure 1-3(a) sont activés par un vecteur d'entrée adéquat. Ainsi, l'impulsion transitoire peut arriver au niveau des entrées D1, D2 et D3 des bascules en fonction de la longueur du chemin de propagation correspondant. Si elle arrive à l'entrée d'une bascule au moment du front actif de l'horloge du système, elle sera capturée et mémorisée comme une valeur erronée (cas des entrées D1 et D3 dans la Figure 1-3(b)).

Dans d'autres cas, l'impulsion transitoire peut être atténuée avant son arrivée aux entrées des bascules. Si sa durée est plus grande que les temps de transition des portes logiques, elle peut se propager aux entrées des bascules sans atténuation [M.B. 97]. Comme les portes logiques deviennent très rapides dans les nouvelles technologies, avec des temps de propagation très courts, les impulsions transitoires ne seront plus atténuées, même pour des particules d'énergie réduite [M.N. 99]. Néanmoins, la durée de l'impulsion finale pourrait être réduite ou augmentée selon les portes traversées.

1.2.1.4. Attaque par laser

Récemment, une nouvelle menace suscite beaucoup d'attention. Elle concerne les circuits relatifs à la sécurité, comme par exemple les primitives cryptographiques ou les cartes à puce. Il a été démontré que des attaques basées sur l'introduction de fautes peuvent permettre à un intrus d'accéder à des informations confidentielles (par exemple une clé secrète) ou de contourner des mécanismes de sécurité. Plusieurs techniques peuvent être utilisées pour générer des fautes dans un circuit, mais la différence la plus significative avec des fautes "naturelles" est principalement la multiplicité spatiale (et potentiellement temporelle) [R.L. 04]. Par exemple, dans un circuit récent, les fautes induites par un laser se limitent difficilement à une seule cellule, à moins que l'attaquant soit équipé d'un matériel très coûteux. Les menaces principales viennent donc des attaques induisant des fautes dans une zone limitée (puisque des attaques globales sur le circuit peuvent souvent être détectées par des capteurs), mais ces attaques ne se limitent pas à l'inversion d'un seul bit, elles peuvent provoquer des inversions multiples.

1.2.1.5. Intégrité du signal

Il est à mentionner que des fautes multiples peuvent également être induites par des problèmes d'intégrité du signal provoquant des fautes intermittentes dans la logique combinatoire. Il est très difficile d'identifier de telles fautes sachant que leur probabilité augmente avec l'évolution des technologies. Prendre en compte l'effet d'erreurs multiples peut donc aider à améliorer la robustesse du circuit par rapport aux limitations de la conception ou de la fabrication.

1.2.1.6. Modélisation des fautes transitoires

Les grandes compagnies du domaine des semiconducteurs et de l'informatique se préoccupent de plus en plus de tous les aspects liés à la sûreté de fonctionnement. Selon Intel, les erreurs transitoires dues aux particules cosmiques (appelées "**soft errors**") deviennent le deuxième grand problème après l'augmentation des courants de fuites dans le domaine sous-micronique [A.C. 99]. La plupart des compagnies observent une augmentation importante des erreurs "softs" à partir de la technologie 0,25 μm . Ces erreurs "softs" correspondent à l'ensemble des cas conduisant à des bits erronés dans le circuit sans effet destructeur (SEU, MBU ou propagation et mémorisation de SET).

Les raisons principales de l'augmentation de la sensibilité sont la réduction de la tension d'alimentation (V_{dd}) et la diminution de la taille du transistor, qui entraîne une réduction de la capacité des nœuds du circuit. Ainsi, la charge déposée par une particule cosmique lors de son impact sur le silicium peut plus facilement inverser la valeur logique associée au nœud. Un tel renversement, s'il affecte un nœud d'une cellule mémoire, peut basculer son état logique, résultant en une erreur SEU. Les mémoires sont les parties considérées comme les plus sensibles aux phénomènes de type SEU et MBU. Pour cette raison, les techniques de durcissement par

conception visent principalement à protéger les mémoires et, par extension, les autres points de mémorisation.

Néanmoins, la miniaturisation des circuits entraîne également une croissance de la sensibilité des parties logiques combinatoires [D.A. 02], [M.S. 03]. Les impulsions transitoires ne seront alors plus filtrées par les portes logiques et se propageront sans encombre jusqu'aux entrées de bascules ou de verrous où elles sont susceptibles d'être capturées. Cette mémorisation a lieu si l'arrivée de l'impulsion au niveau de l'entrée d'un point mémoire coïncide avec le front actif de l'horloge. La fréquence d'horloge des systèmes devenant de plus en plus grande, la probabilité de capture augmente proportionnellement. Les SETs représenteront donc une part croissante des erreurs "softs", avec pour conséquence une augmentation des erreurs multiples (plusieurs bits erronés simultanément générés).

- **Extension au modèle de fautes MBF**

Il existe différents niveaux de modélisation du phénomène "soft errors", allant d'un modèle spécifique aux descriptions de circuit de très bas niveau (netlist niveau transistor ou descriptions physiques) jusqu'à des modèles utilisés dans des descriptions de circuit niveau porte ou niveau RTL.

Dans notre étude, nous nous sommes focalisés sur la modélisation des fautes au niveau RTL. A ce niveau, les registres de données et les registres d'état sont facilement identifiables. Les modèles de fautes associés aux phénomènes SEU/MBU correspondent respectivement à une inversion de bit unique ou multiple dans de tels registres. Par contre, il est souvent impossible de modéliser avec précision le phénomène SET car, à ce niveau de description, les nœuds d'une logique combinatoire ne sont pas identifiables. Il est donc impossible d'évaluer avec précision les configurations d'erreur dues aux SETs qui peuvent éventuellement se produire. Cependant, supposer que seulement des inversions de bits uniques peuvent se produire est évidemment très optimiste. En conséquence, des modes de défaillance critiques peuvent ne pas être identifiés pendant cette analyse haut niveau et devraient donc être traités beaucoup plus tard dans le processus de conception. Une solution à cette limitation consiste à étendre le modèle de fautes habituel à un modèle que nous appelons **MBF** (multiple bit-flips). En effet, une impulsion transitoire qui se propage à travers des portes logiques sera finalement capturée par des éléments mémoires. Cela correspond à une inversion de bits multiple au niveau des points mémoires.

Par ailleurs, lorsqu'on considère une attaque par laser, les erreurs peuvent ne pas se limiter à un seul bit. Par conséquent, analyser la sensibilité d'un circuit donné face aux attaques qui touchent la sécurité implique de tenir compte des MBFs, au moins sur les emplacements les plus critiques.

Une autre source de fautes peut être prise en considération par le modèle MBF. Elle est liée au problème d'intégrité du signal. En effet, dans ce cas, des erreurs simultanées de bit peuvent également se produire dans plusieurs registres qui n'ont aucune relation fonctionnelle particulière.

- **Transitions erronées**

Un cas particulier consiste à modéliser à un niveau élevé l'effet d'une erreur "soft" qui survient dans un registre d'états défini par une variable dans un type énuméré, utilisant des noms symboliques. A ce niveau, la spécification niveau bit n'est pas encore disponible avant la synthèse (c.-à-d., avant l'affectation des codes binaires aux noms symboliques d'état). Fondamentalement, l'effet d'une inversion de bit dans un registre d'états est une modification de l'état actuel du circuit. La solution proposée est donc d'injecter des transitions erronées entre les états (elle peut même être appliquée aux descriptions de haut niveau qui ne sont pas synthétisables). Le modèle d'erreur "transitions erronées" présenté dans la section 3.2 correspond à un modèle plus abstrait où l'occurrence d'un SEU dans un registre assigné avec des noms symboliques est assimilée à une transition inattendue d'une valeur symbolique à une autre. Puisque les codes binaires ne sont pas encore connus, il n'est pas possible de limiter les transitions erronées à des cas correspondant à des inversions de bits uniques. C'est donc un cas particulier où les erreurs d'inversion multiple de bits (MBF) sont prises en considération.

Ces modèles de haut niveau "transitions erronées" et SEU/MBF sont proposés dans le cadre de la thèse et seront détaillés dans les chapitres 2, 3 et 4.

Deux approches permettent de diminuer la sensibilité aux rayonnements. L'une, au niveau technologique, utilise des méthodes de fabrication de composants plus robustes ; l'autre agit soit au niveau de la conception du système entier où le durcissement se réalise plutôt grâce à des architectures tolérantes aux fautes transitoires, soit par la modification de la taille des dispositifs sensibles aux SEE (tels les registres, les points mémoires, ...), soit par diverses techniques de protection à base de redondance spatiale, temporelle ou d'information.

Les deux moyens de prévention (technologique et architectural) ne permettent pas toujours une protection totale face aux erreurs produites par les upsets. Par ailleurs, les applications commerciales ne peuvent pas se permettre un coût très élevé de conception et de processus technologique. Il est donc indispensable de pouvoir évaluer la sensibilité d'un circuit et de décider du niveau de protection nécessaire pour ne pas augmenter le coût de durcissement de façon injustifiée.

Pour ce faire, des simulations de l'impact d'une particule énergétique sur le comportement du circuit avec des calculs de prédiction sont nécessaires. Cela peut être réalisé en modélisant la faute étudiée à un niveau d'abstraction correspondant au niveau de description du circuit cible (niveau physique, électrique ou comportemental). Il devient alors possible de prévoir les actions adéquates pour éliminer ou diminuer les risques de défaillances lorsque ces systèmes fonctionnent dans l'espace ou dans des environnements à risques.

Cette évaluation, ainsi que l'analyse de l'efficacité des méthodes de durcissement implémentées, nécessite d'observer la réponse du circuit en présence de fautes et conduit donc à définir des méthodes **d'injection de fautes**. Ces méthodes feront l'objet des sections suivantes de ce chapitre.

1.2.2. Fautes de type logiciel

Les fautes de type logiciel sont toujours la conséquence d'une conception incorrecte, que se soit au niveau de la spécification ou au niveau du codage. Tous les ingénieurs de développement logiciel savent qu'un produit logiciel demeure sans bogues jusqu'à ce que le prochain bogue soit trouvé. Plusieurs de ces fautes sont latentes dans le code et apparaissent seulement pendant l'exécution et plus particulièrement sous des programmes d'essai (workload) lourds ou peu communs ou sous des contraintes de synchronisation.

Puisqu'ils sont le résultat d'une mauvaise conception, on pourrait supposer que toutes les fautes logicielles soient permanentes. Cependant, la pratique montre qu'en dépit de leur nature permanente, leur comportement est transitoire, c'est-à-dire, quand un mauvais comportement du système se produit, il peut ne pas être observé une deuxième fois, même si l'on prend un grand soin pour répéter la situation dans laquelle il s'est produit. Un tel comportement s'appelle généralement un échec du système. Les subtilités de l'état du système peuvent masquer la faute, comme lorsqu'un bogue est déclenché par une synchronisation très particulière entre plusieurs composants du système, ou par d'autres situations rares et non reproductibles.

Pendant le processus global du développement d'un logiciel, des fautes peuvent être créées dans chaque étape.

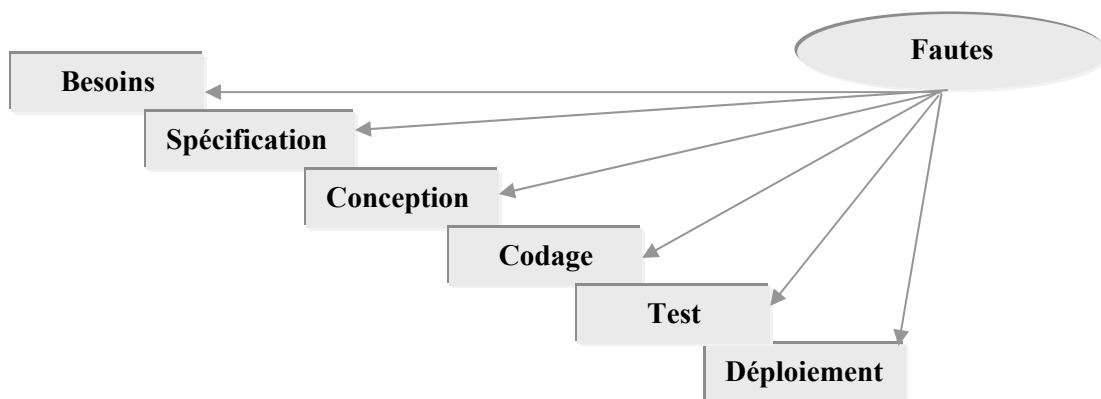


Figure 1-4: Génération de fautes pendant le procédé de développement d'un logiciel

Ces fautes peuvent être cataloguées comme suit :

- **Fautes au niveau fonctionnel** : implémentation incorrecte ou absente, ce qui exige des changements au niveau de la conception.
- **Fautes au niveau algorithmique** : implémentation incorrecte ou absente, qui peut être corrigée sans avoir à changer de conception.
- **Fautes au niveau de la sérialisation** : sérialisation absente ou incorrecte des ressources partagées.
- **Fautes de vérification** : validation incorrecte ou absente des données, boucle incorrecte, ou instruction conditionnelle incorrect.

- **Fautes d'assignement** : les valeurs sont mal ou pas assignées.

Curieusement, la plupart des défaillances d'un système ont été souvent affectées soit à des fautes de type logiciel, soit à des fautes permanentes de type matériel, en tenant peu compte des fautes matérielles de types transitoire ou intermittent. Plusieurs études ont cependant montré que ces types de fautes sont beaucoup plus fréquents que les fautes permanentes. Le problème est qu'il est beaucoup plus difficile de les dépister. C'est ce type de fautes qui sera considéré dans la suite du document.

1.3. Environnement d'injection de fautes

Durant notre travail de thèse, nous nous sommes intéressés à la modélisation à haut niveau des effets d'upsets dans les points mémoires d'un circuit. L'objectif consiste à évaluer la vulnérabilité des architectures digitales face aux effets des radiations résultant en l'inversion de bits dans des cellules de mémorisation. Pour ce faire, nous avons proposé de nouvelles méthodologies d'injection de fautes qui seront utilisées pour obtenir des données expérimentales sur le comportement du circuit consécutif aux fautes induites par les upsets.

Au début des années 70, des techniques basées sur l'injection de fautes ont été utilisées dans l'industrie pour mesurer la fiabilité des ordinateurs. Au milieu des années 80, le milieu universitaire a commencé activement à utiliser l'injection de fautes pour conduire la recherche expérimentale. Le travail initial s'est focalisé sur la compréhension de la propagation d'erreurs et l'analyse de l'efficacité de nouveaux mécanismes de durcissement. L'injection de fautes cible aussi les systèmes répartis, et même de nos jours, l'Internet. En outre, les diverses couches du système informatique s'étendant du matériel au logiciel peuvent être visées par l'injection de fautes. Les sections suivantes se limitent aux principales approches d'injection ayant été appliquées sur des circuits numériques.

La Figure 1-5 présente une vue globale d'un environnement d'injection de fautes. Il inclut un système cible, un injecteur de fautes associé à une bibliothèque de fautes, un générateur de stimuli ("workload") associé éventuellement à une bibliothèque, un contrôleur, un moniteur, un collecteur de données et un analyseur de données.

Pendant l'exécution des entrées fonctionnelles obtenues du générateur de "stimuli" (applications, benchmarks, ou stimuli synthétiques), l'injecteur de fautes active les fautes dans le système cible à des instants spécifiés par le concepteur. Le rôle du moniteur consiste à observer l'exécution des commandes et à initier l'enregistrement des données lorsque c'est nécessaire. Le collecteur de données effectue l'enregistrement des données en temps réel (en ligne), tandis que l'analyseur, qui effectue le traitement des données, est initié dans certains environnements en différé (hors ligne). Le contrôleur se charge de commander l'ensemble des expériences d'injection.

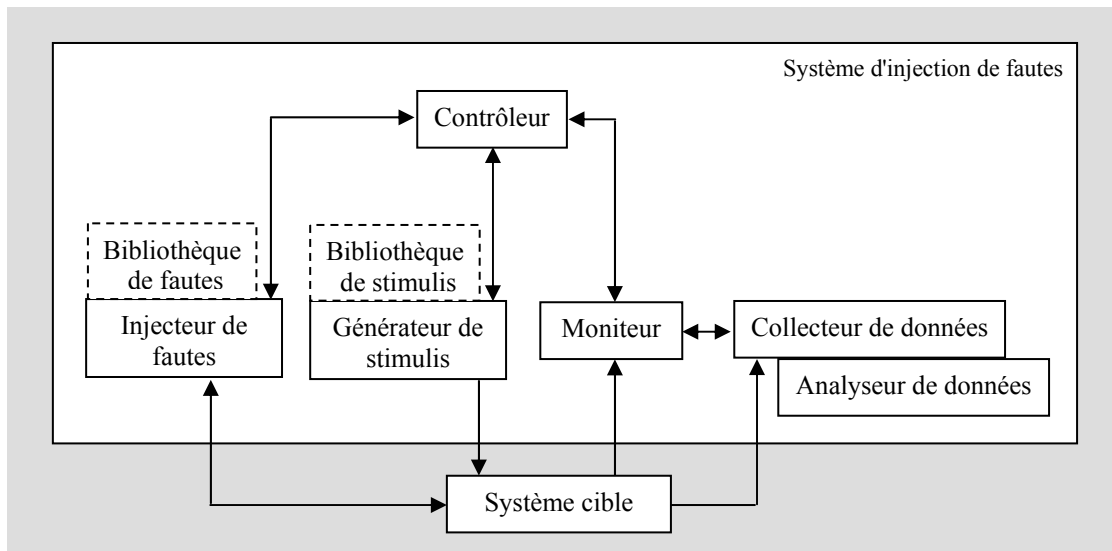


Figure 1-5: Les composants de base d'un environnement d'injection de fautes

Dans la pratique, le contrôleur est un programme qui peut s'exécuter soit sur le système cible soit séparément sur un ordinateur. L'injecteur de fautes peut correspondre à une implémentation matérielle ou logicielle. Il peut supporter différents types de fautes (avec une sémantique matérielle ou une structure logicielle appropriée), injectées à des emplacements multiples et à des instants différents. Ces valeurs sont extraites de la bibliothèque de fautes. La bibliothèque de fautes, le générateur de stimuli, le moniteur ainsi que les autres composants de la Figure 1-5 permettent, lorsqu'ils sont indépendants, d'obtenir une grande flexibilité et portabilité.

1.4. Injection de fautes basée sur le matériel

Un grand nombre de méthodes proposées se basent essentiellement sur la validation physique des systèmes. Ces méthodes incluent des injections sur les broches des circuits, des corruptions en mémoire [T.M. 94], des injections d'ions lourds, des perturbations des alimentations [J.K. 91], ou encore l'injection de fautes par laser [J.R. 97]. Aucune de ces approches ne peut être utilisée pour une évaluation de la sûreté avant que le circuit ne soit effectivement fabriqué.

On distingue deux types de techniques d'injection de fautes basées sur le matériel :

- **Avec contact** : L'injecteur possède un contact physique direct avec le système cible, il exerce un contrôle externe sur les cibles d'injection en appliquant des changements de voltage ou de courant. Par exemple, les méthodes de forçage et d'insertion au niveau broches sont de ce type.
- **Sans contact** : L'injecteur ne possède pas de contact physique direct avec le système cible. Cependant, une source externe produit un phénomène physique tel que l'émission de particules ou d'interférences électromagnétiques, ce qui permet de générer un courant parasite à l'intérieur du circuit.

Les injections de fautes matérielles ciblent des exemples réels de circuit après fabrication. Le circuit est soumis à une certaine interférence afin d'engendrer la faute et le comportement résultant est examiné. Jusqu'ici, les travaux réalisés concernent essentiellement des fautes d'origine externe (permanentes ou transitoires). Par ailleurs, pour certaines techniques, il est impossible d'injecter des fautes permanentes de type "collage" et "court-circuit". La mise en œuvre des expériences nécessite du temps pour préparer l'environnement de test du circuit. Mais généralement ce genre de test s'exécute plus rapidement qu'une simulation, ce qui permet au concepteur de lancer des applications plus complexes et par conséquent d'obtenir une analyse comportementale plus réaliste du circuit en présence de fautes. Le test est utilisé pour valider le circuit juste avant ou pendant sa production. Ces techniques ne sont pas intrusives dans la mesure où le comportement original du circuit n'est pas altéré.

1.4.1. Injection de fautes niveau broches

L'injection de fautes basée sur le matériel peut utiliser un contact direct avec les broches du circuit. Elle est alors appelée "injection niveau broches" et a été la méthode d'injection basée sur le matériel la plus utilisée. Il existe deux techniques principales pour altérer les courants électriques et les voltages au niveau des broches :

- **Technique de forçage** : Cette technique permet d'introduire un courant additionnel via des pointes (appelés sondes) connectées aux broches du circuit afin d'altérer les courants électriques et forcer les points choisis à un niveau logique bas ou haut. Cette méthode est généralement limitée aux fautes de collage, bien qu'il soit possible d'obtenir des fautes de type "court circuit" en reliant à l'aide d'une sonde deux broches ou plus. Cette technique qui consiste à forcer un courant additionnel doit être appliquée avec précaution. Une quantité excessive de courant peut endommager le matériel cible.
- **Technique d'insertion** : Elle consiste à insérer un dispositif d'injection de fautes (appelé "socket" en anglais) entre le circuit cible et la carte de test. Le dispositif inséré peut injecter sur les broches du circuit cible des fautes de type collage, court-circuit ou des fautes logiques plus complexes. Les signaux d'une broche peuvent être inversés (par un NON logique) ou associés aux signaux d'une broche adjacente par l'intermédiaire de portes logiques AND/OR. Ils peuvent également être associés aux valeurs précédentes du signal de la même broche. La technique d'insertion ne présente aucun danger d'endommagement par rapport aux composants étudiés.

Les deux méthodes vues précédemment offrent une bonne contrôlabilité temporelle et spatiale de l'injection avec une perturbation du système cible très limitée voire inexistante. Toutefois, les fautes modélisées au niveau broches ne sont pas tout à fait identiques aux modèles de fautes traditionnels car elles ne peuvent se produire à l'intérieur du circuit.

Parmi les outils d'injection développés dans le domaine de la recherche universitaire nous pouvons citer notamment :

- **Messaline**

Messaline est un système de forçage de fautes au niveau broches développé au LAAS-CNRS, France [J.A. 91, J.A. 90]. La faute injectée est directement appliquée sur l'une des broches du circuit intégré cible, et il est possible de contrôler sa durée et sa fréquence. Les modèles de fautes supportés incluent les fautes de type collage transitoires (qui affectent des broches uniques ou multiples), de type court-circuit ou des fautes logiques plus complexes.

Messaline est composé de quatre modules : Module d'Injection, Module d'Activation, Module de Collection et Module de Gestion. Le module d'injection peut cibler simultanément jusqu'à 32 points d'injection en utilisant les deux techniques d'injection : forçage et insertion. Le module d'activation assure l'initialisation du système cible. Le module de collection et d'affichage est utilisé pour collecter le contenu des éléments observés (les signaux d'erreur, les sorties du système, etc.). Le module de gestion est responsable de la génération automatique et paramétrable de la séquence de test, du contrôle du temps d'exécution et de l'archivage du résultat pour une analyse post-test.

Messaline a notamment été employé pour la validation expérimentale de deux systèmes : 1) un sous-ensemble d'un système d'enclenchement automatisé et centralisé pour les applications de contrôle ferroviaire, et 2) un système réparti qui correspond à une implémentation courante d'un système de communication fiable issu du projet "ESPRIT Delta-4" [D.P. 94].

- **RIFLE**

RIFLE est un système d'injection de fautes au niveau broches pour la validation de la fiabilité [H.M. 1994]. Il a été développé à l'université de Coimbra, Portugal. Ce système peut être adapté à un éventail de systèmes à base de microprocesseur. Les fautes sont principalement injectées dans les broches du processeur. RIFLE est doté d'un module original dérivé des analyseurs logiques : signal de déclenchement et module de mémorisation de "traces", qui supprime l'usage des mécanismes de retour d'information, lents, présents sur les autres systèmes. La durée des tests s'en trouve raccourcie.

L'injection de fautes est déterministe et peut être reproduite si nécessaire. Elle se pratique par forçage et insertion. Les modèles de fautes sont pratiquement les mêmes que ceux de Messaline. La durée des fautes est paramétrable permettant ainsi l'injection de fautes permanentes ou transitoires. RIFLE peut également détecter des circonstances spécifiques dans lesquelles les fautes injectées n'affectent pas le système cible.

[H.M. 1994] présente des résultats de couverture de fautes et de latence obtenus avec RIFLE pour un ensemble de mécanismes de détection d'erreur simple.

1.4.2. Injection par particules

Des fautes peuvent aussi être injectées par l'émission d'ions lourds ou d'autres particules créés par un matériel dédié. Cette technique est appréciée par le fait qu'elle imite fidèlement les phénomènes physiques observés notamment dans l'espace. L'intérêt aussi de l'injection de fautes par des ions lourds est que les fautes injectées dans des circuits peuvent cibler des emplacements qui sont difficiles (et souvent impossibles) à atteindre par d'autres techniques d'injection. Par contre, il est difficile d'obtenir un bon contrôle spatial ou temporel. En effet, les fautes transitoires engendrées dans le circuit sont généralement réparties suivant une distribution aléatoire.

Les expériences d'injection de fautes par bombardement de particules sont la plupart du temps réalisées avec des équipements lourds de type accélérateur de particules, tels que ceux de Los Alamos National Laboratory ou de Louvain La Neuve [R.V. 98]. Des équipements plus légers peuvent aussi être employés, basés sur des sources radioactives (ions lourds ou particules alpha).

Des expériences d'injection de fautes basées sur le rayonnement d'ions lourds (HI) ont été par exemple effectuées à l'université de Technologie de Chalmers, Göteborg, Suède [U.G. 89, J.K. 94]. Un rayonnement d'ions lourds d'une source **Californium-252** a été employé pour injecter des SEUs à des endroits internes aux circuits intégrés. Cela nécessite l'utilisation d'une mini chambre à vide pour loger la source de californium et le circuit à tester dont le boîtier doit être retiré. Les broches du circuit intégré cible sont prolongées et traversent la base de la mini chambre de telle sorte qu'il soit branché directement à la carte. De plus un deuxième circuit de référence doit être utilisé pour comparer le fonctionnement normal au fonctionnement perturbé. La chambre à vide contient un obturateur commandé électriquement qui est utilisé pendant l'amorçage pour protéger le circuit sous test du rayonnement.

Bien que les ions lourds émis de la source Cf-252 ne provoquent pas une parfaite imitation de l'impact des ions lourds dans l'espace ou du rayonnement neutronique sur terre (exemple : le rapport entre l'inversion de bits multiple et unique diffère), la méthode fournit une approche pratique pour évaluer l'efficacité des mécanismes de détection et de recouvrement d'erreur lorsqu'il s'agit de fautes de type inversion de bits. Exposer des circuits au rayonnement neutronique est moins pratique puisqu'il implique de placer le système à tester dans une chambre dotée d'une armature en béton [P.H. 00].

Pour le CPU 68070, les ions lourds du Cf-252 provoquent principalement des inversions de bits uniques; le pourcentage des erreurs de bits multiples induites dans les registres principaux est inférieur à 1%. Ces expériences sont rapportées dans [R.J. 94]. La méthode d'ion lourd a été précédemment utilisée pour évaluer plusieurs mécanismes de détection d'erreur implémentés en matériel/logiciel pour le microprocesseur MC6809E.

Développé à l'université de Vienne-Autriche [E.F. 96-a], **MARS (MAintainable Real-time System)** est un système réparti, temps réel et tolérant aux fautes. Des mécanismes de détection

d'erreurs sont implémentés à trois niveaux: matériel, logiciel et le niveau application. Les chercheurs ont évalué l'efficacité de ces dispositifs de détection d'erreurs en utilisant trois méthodes d'injection : injection niveau broches, émission d'ions lourds mais aussi interférences électromagnétiques pour réaliser des injections de fautes sans contact. Les résultats montrent que les trois méthodes sont complémentaires car elles génèrent différents types d'erreur. Les méthodes d'injection par émission d'ions lourds et par interférence électromagnétique sont les mieux adaptées lorsqu'il s'agit de tester les mécanismes de détection d'erreurs au niveau logiciel et application.

1.4.3. Injection de fautes par laser

Une approche alternative au test par radiation consiste à réaliser de l'injection de charge avec un laser pulsé. En effet, dans un mode non destructif, un laser pulsé peut permettre d'obtenir des informations spatiales et temporelles sur les régions sensibles et sur la réponse d'un circuit cible aux fautes de type SEE (ce qui est difficile avec un rayonnement de particules). Le laser pulsé peut également être utilisé pour vérifier, avant de passer à une source de rayonnement, que l'installation expérimentale de test fonctionne correctement.

Les techniques d'excitation par laser permettent d'irradier avec précision des zones de systèmes complexes (Processeur : cache, registres, pipeline d'un processeur, logique de prédiction de branchement, etc... DRAM : Logique de contrôle, zone mémoire, etc..)

Les équipes de recherche qui ont effectué un travail étendu dans l'injection de SEEs basé sur le laser pulsé sont peu nombreuses. Nous pouvons citer celles du laboratoire naval de recherche (Washington, USA), de la société Aerospace Corporation (Los Angeles, USA), de l'Université de Bordeaux (France) et de MBDAUK Ltd (Royaume-Uni). Nous ne parlerons pas ici de l'utilisation de ces techniques pour réaliser des attaques par fautes sur des circuits utilisés dans des applications sécuritaires.

Un groupe de chercheurs du laboratoire IXL (Bordeaux, France) a mis en place un banc laser pulsé qui permet de faire des tests non destructifs sur des circuits intégrés en fonctionnement. Des impulsions laser ultra courtes (femtoseconde, picoseconde) sont utilisées pour perturber localement, avec une résolution sub-micronique (adaptée à la taille des circuits intégrés de dernière génération), le circuit intégré en fonctionnement. La détection des effets de cette perturbation s'effectue au niveau des grandeurs électriques accessibles à l'extérieur du circuit. L'automatisation du banc permet de réaliser des cartographies basées sur l'évolution de nombreux paramètres de l'onde électrique (temps de montée, valeur maximale, valeur crête à crête, etc...) avec une résolution spatiale inférieure au micron. La précision temporelle de l'impulsion laser permet de mettre en œuvre des études dynamiques. Diverses applications ont été développées.

Dans le domaine de l'analyse des défaillances, les possibilités de simuler expérimentalement l'effet d'un SEU à l'aide d'un laser pulsé font l'objet d'un intérêt particulier [H.H. 65, S.B. 96, V.P. 99]. Pendant les deux dernières décennies, des effets de rayonnement sur

des cellules de SRAM ont été largement étudiés [J.Z. 87, Y.M. 95, P.D. 96]. Ce travail a été motivé par le fait qu'un équipement de rayon laser est un outil puissant capable de créer des SEEs à des endroits précis dans un circuit intégré. D'ailleurs, cela a permis aussi de comprendre les mécanismes de base responsables des SEUs dans les mémoires statiques.

Dans [F.D. 02], il a été démontré que non seulement les drains des transistors bloqués sont sensibles, comme cela a été indiqué beaucoup de fois [F.S. 96, P.R. 99], mais également d'autres régions de la cellule comme les transistors d'activation de lecture/écriture. [F.D. 02] propose une étude systématique de la sensibilité d'une cellule simple de SRAM au phénomène SEU en balayant en détail le dessin des masques complet d'une cellule élémentaire. Le but est alors de préciser et d'expliquer la présence de zones sensibles inattendues qui augmentent de manière significative la sensibilité globale.

Le dispositif d'excitation laser est décrit en détail dans [J.M. 94, S.B. 96]. Ces articles mentionnent aussi les caractéristiques les plus importantes de la technique d'injection de SETs. Le laser émet des impulsions de lumière courtes qui peuvent être focalisées vers le bas à l'aide d'un petit spot positionné sur n'importe quel transistor dans le circuit afin de déterminer si celui-ci est sensible aux SETs. Chaque dispositif de laser pulsé possède une configuration laser et optique unique. Cependant, tous ces dispositifs partagent un certain nombre d'attributs communs qui incluent une longueur courte de pulsation lumineuse (d'environ 1 picoseconde) et une petite taille de spot (entre 1 et 2 microns de diamètre). La longueur d'onde de la lumière détermine la profondeur de sa pénétration dans le silicium ; une lumière avec une longueur d'onde de 800 nm a une profondeur de pénétration dans le silicium d'environ 15 μ m. Une lumière avec une longueur d'onde plus courte a également une profondeur de pénétration plus courte.

Lors d'une expérimentation, le circuit cible est fixé sur une station mobile "X-Y". Un objectif microscopique (100X) est utilisé pour focaliser la lumière. Le circuit est déplacé à l'aide de la station "X-Y" afin de focaliser le spot lumineux sur un transistor sensible aux SETs. En changeant l'intensité de la lumière à l'aide de filtres de densité, la dépendance de l'amplitude d'un SET par rapport à la charge déposée (qui est proportionnelle à l'énergie de l'impulsion laser) peut être mesurée. Par ailleurs, mesurer la durée d'une impulsion pour établir le flux maximum d'ion peut prévenir contre le débordement de SET particulier appelé : SET "longue-durée" [C.P. 03].

Dans [S.B. 01] des changements de mesure dans la forme d'impulsion d'un SET généré dans des transistors spécifiques de l'amplificateur "LM119" provoquent différents niveaux de défaillance. L'identification des différentes formes d'impulsion obtenues dans les différentes régions sensibles de l'amplificateur LM139 [R.K. 00, C.P. 01] et LM124 [C.P. 01, R.P. 02, R.M. 01, P.A. 00] permet d'évaluer l'impact des fautes transitoires sur une application donnée.

1.4.4. Avantages et inconvénients

Les approches d'injection de fautes basées sur le matériel offrent plusieurs possibilités. **Les principaux avantages** sont :

- Certaines techniques d'injection peuvent cibler des emplacements qui sont difficiles d'accès par d'autres moyens. Par exemple, la méthode basée sur l'émission de particules peut cibler des endroits dans un circuit VLSI qui sont impossibles à atteindre avec d'autres méthodes.
- Elles sont non intrusives : aucune modification du circuit n'est nécessaire pour l'injection de fautes.
- Les expériences d'injection de fautes sont rapides et peuvent être exécutées à fréquence nominale ou proche, ce qui permet d'exécuter un grand nombre d'expériences d'injection de fautes.
- Elles sont réalisées sur un circuit réel qui peut exécuter l'application utilisée dans son environnement opérationnel.
- Elles sont bien adaptées pour un système qui a besoin d'une résolution temporelle élevée.
- Elles sont plus adaptées pour imiter des fautes réelles qui peuvent survenir dans l'application.
- L'évaluation expérimentale par l'injection dans un matériel réel est, dans beaucoup de cas, la seule manière pratique d'estimer exactement la couverture et la latence.
- Les techniques d'injection sans contact ont une faible perturbation.

Par ailleurs, **les principaux inconvénients** sont :

- L'injection de fautes peut présenter un grand risque d'endommagement du système cible.
- Le niveau élevé d'intégration, circuit hybride multi-puce et packaging technologique dense limitent l'accessibilité à l'injection.
- Les méthodes exigent un matériel spécifique pour exécuter les expériences d'injection de fautes. En général, ce matériel peut être difficile et coûteux à obtenir.
- L'observabilité et la contrôlabilité peuvent être limitées.
- Quelques méthodes d'injection de fautes niveau matériel, telles que la mutation d'état, exigent l'arrêt et l'initialisation du processeur pour injecter une faute, elle n'est pas toujours efficace pour mesurer les latences dans les systèmes physiques.
- L'injection de fautes au niveau broches est limité au niveau des cibles.
- L'injection par particules se limite aux fautes transitoires et il n'est pas possible de maîtriser leur taux d'occurrence, ou leur localisation.

1.5. Injection de fautes basée sur le logiciel

1.5.1. Méthodes et outils

Ces dernières années, les chercheurs montrent plus d'intérêt pour le développement d'outils d'injection de fautes basée sur le logiciel (Software Implemented Fault Injection : SWIFI). Cette technique est attractive car elle permet de réduire le coût et de fournir une alternative de contrôle plus facile comparé aux autres techniques d'injection qui exigent un équipement matériel spécial et des interfaces spécifiques au système cible. Par ailleurs, une injection de type SWIFI vise souvent l'application et éventuellement le système d'exploitation qui s'exécute sur le système cible, ce qui est difficile à effectuer par les techniques d'injection basées sur le matériel.

Il s'agit donc de modifier l'état du système selon le modèle conçu par le programmeur. Typiquement, cela correspond à changer le contenu de la mémoire ou des registres en se basant sur des modèles de fautes spécifiques afin d'émuler les conséquences de fautes matérielles ou pour injecter des fautes logicielles (les bogues cachés) [J.C. 98].

Si la cible est une application, l'injecteur de fautes sera inséré soit dans l'application elle-même, soit entre l'application et le système d'exploitation. Cependant, si la cible est un système d'exploitation, l'injecteur de fautes doit être embarqué dans le système d'exploitation car il est très difficile d'ajouter une couche entre le matériel et le système d'exploitation.

Généralement, on cible le programme qui a des fonctions communicatives ou coopératives de sorte qu'il y ait assez d'interaction pour rendre l'injection de fautes utile. Toutes les sortes de fautes peuvent être injectées, allant des fautes dans les registres et les mémoires, aux paquets réseaux, ou aux erreurs liées aux conditions de vérification d'erreur et drapeaux. Ces fautes peuvent aussi être injectées lors des simulations de systèmes complexes, là où l'on maîtrise les interactions mais pas les détails de l'implémentation.

Nous pouvons faire la distinction entre les méthodes d'injection basées sur le logiciel en se basant sur l'instant où les fautes sont injectées : pendant la compilation ou pendant l'exécution.

- **Injection lors de la compilation** : L'approche consiste à injecter des fautes au niveau du code machine et ensuite charger l'application mutée (segment de code ou segment de données) dans le système cible. La modification des instructions du programme est réalisée avant que l'image du programme ne soit chargée en mémoire et exécutée. Au lieu d'introduire des fautes dans le matériel d'un système cible, cette méthode injecte des erreurs dans le code source ou dans le code assembleur du programme cible afin d'émuler l'effet matériel, logiciel et transitoire des fautes. L'injection génère une image logicielle erronée et c'est lorsque que le système exécute cette image fautive que la faute est activée. Le code modifié altère les instructions du programme cible et provoque l'injection.

Puisque l'effet de la faute est introduit dans du code machine, les concepteurs peuvent utiliser cette méthode pour émuler des fautes permanentes. La méthode est simple à implémenter mais elle limite les possibilités d'injection et ne permet pas d'effectuer des injections de fautes pendant l'exécution du programme (les injections sont statiques puisqu'elles sont déterminées avant l'exécution).

Trois raisons principales peuvent amener à choisir une telle approche [E.F. 96-b] :

- 1) L'intrusion est réduite au minimum puisqu'il s'agit uniquement de modifier le programme qui évalue l'effet de la faute (pas besoin d'activer un programme additionnel pendant l'exécution). En outre, elle ne cause aucune perturbation du système cible pendant l'exécution.
 - 2) L'injection de fautes au niveau du code machine permet d'injecter des fautes qui ne peuvent pas être injectées à des niveaux plus élevés en utilisant des mutations de code source [Y.C. 98, J.V. 98].
 - 3) L'injection au niveau compilation s'intègre facilement avec le processus de développement d'application car les applications sont développées, configurées, assignées, et programmées hors ligne sur un ordinateur principal (host) et ensuite chargées sur le système cible.
- **Injection lors de l'exécution** : Pendant l'exécution, il est nécessaire d'introduire un mécanisme pour activer l'injection de fautes. Les mécanismes d'activation communément utilisés incluent :
 - **Temps écoulé** : Dans cette technique qui est considérée comme la plus simple, on utilise un temporisateur qui expire après un temps prédéterminé et la faute est déclenchée. En effet, une fois le temps écoulé, une interruption est générée afin d'invoquer l'injection de fautes. Le temporisateur peut correspondre à une implémentation matérielle ou logicielle. Cette méthode n'implique aucune modification de l'application ou du programme "stimulis". Un temporisateur matériel doit être lié au vecteur d'interruption système. Etant donné que les injections se basent sur la temporisation plutôt que sur des événements spécifiques ou sur l'état du système, cela rend imprévisible les effets d'une faute et par conséquent le comportement du programme. Ceci convient particulièrement à la modélisation des fautes transitoires et intermittentes de type matériel.
 - **Exception/Interruption** : Dans ce cas, une interruption matérielle ou une exception logicielle transmet le contrôle à l'injecteur de fautes. Contrairement à la méthode basée sur la temporisation, la méthode "Exception/Interruption" permet d'injecter une faute dès qu'un événement ou une condition se présente. Par exemple, une instruction d'exception logicielle insérée dans le programme cible invoquera l'injection de faute avant que le programme n'exécute une instruction particulière. Lors de l'exécution de l'exception, une interruption est générée afin de transférer le contrôle à un gestionnaire d'interruption. Une interruption matérielle invoque l'injection lorsqu'un événement

matériel (qui fait parti des événements observés) survient (par exemple, un accès à un emplacement mémoire particulier). Les deux mécanismes doivent être liés au vecteur de gestion des interruptions.

- **Insertion de code** : Dans cette technique, des instructions sont ajoutées au programme cible. Elles permettent d'exécuter l'injection de fautes avant d'exécuter des instructions particulières. Cela ressemble à la méthode d'injection lors de la compilation sauf que pour l'insertion de code, cela consiste à ajouter des instructions au lieu de modifier les instructions originales et à effectuer l'injection pendant l'exécution.

Contrairement à la méthode d'Exception/Interruption, l'injecteur de fautes est inclus dans le programme cible et s'exécute en mode utilisateur au lieu de s'exécuter en mode système.

Les injections de fautes basées sur le logiciel sont orientées vers les détails d'implémentation, et peuvent adresser aussi bien l'état d'un programme que la communication et les interactions. Les fautes sont de type "temps écoulé", "message absent", "répéter une exécution", mémoire ou registres corrompus, lecture sur disque erronée, et la plupart des autres états d'accès. Le système fonctionne alors en présence de fautes afin d'étudier son comportement. Les injections peuvent être non intrusives, particulièrement si les caractéristiques temporelles essentielles ne sont pas modifiées. Dans le cas contraire, si le mécanisme d'injection nécessite un certain temps pour introduire les fautes, il peut ainsi perturber l'activité du système et causer une incohérence des résultats temporels par rapport au système de référence sans le mécanisme d'injection de fautes. Ceci se produit parce que le mécanisme d'injection fonctionne sur le même système lorsque l'application est testée.

Parmi les outils d'injection développés dans le domaine de la recherche universitaire nous pouvons citer les suivants :

- **FIAT**

Développé à l'université de Carnegie Mellon, FIAT (Fault Injection-based Automated Testing environment) est un environnement automatisé d'injection de fautes temps réel accéléré et distribué [Z.S. 88]. Il est capable d'émuler une variété d'architectures distribuées. Il fournit des possibilités pour contrôler le comportement du système et injecter des fautes afin d'obtenir une caractérisation et une validation expérimentale de la fiabilité d'un système. L'environnement de FIAT génère avec facilité des expérimentateurs pour définir des classes de fautes (rapports entre les fautes et les modèles d'erreur qu'elles causent) et pour indiquer (par exemple, au code source d'une application) où, quand et combien de temps les erreurs vont sévir et comment elles interagissent avec le code objet ou les données en exécution. Dans sa version initiale, le logiciel FIAT peut injecter dans le code d'une application utilisateur et dans des messages (corrompus ou perdus), tâches (retardées et anormales) et temporisateurs. Les versions postérieures ont étendu les possibilités d'injection de fautes dans les systèmes d'exploitation.

- **FERRARI (Fault and Error Automatic Real-time Injector)**

Développé à l'université du Texas à Austin, FERRARI est destiné à évaluer la fiabilité de systèmes complexes par émulation de fautes matérielles [G.K. 92, G.K. 95]. Il utilise des exceptions logicielles pour injecter des fautes dans le CPU, la mémoire et le bus. Il a été implanté sur station de travail Sun 4 Sparc dans un environnement X-Windows. Il se compose de quatre modules :

- **Initialisation et activation** : ce module sert à préparer la cible pour l'injection.
- **Information utilisateur** : permet à l'utilisateur d'introduire des paramètres relatifs à l'expérimentation.
- **Injecteur de fautes/erreurs** : il utilise des exceptions et des procédures de gestion d'exception. Une fois l'exception logicielle activée par un temporisateur, elle exécute la routine pour injecter des fautes à des emplacements spécifiques. Typiquement, le contenu des registres ou des emplacements mémoires sélectionnés est modifié afin d'émuler des corruptions de données réelles. Les fautes injectées peuvent être permanentes ou transitoires; elles peuvent provoquer des erreurs de ligne d'adresse ou de donnée.
- **Collecte et analyse** : récupère des données comportementales, les traite, effectue des calculs de latence et de couverture de fautes et délivre les résultats.

- **DOCTOR**

DOCTOR (Integrate**D** SOftware Fault Inje**CT**iOn Envi**R**onment), développé à l'université du Michigan et décrit dans [S.H. 93], permet de cibler la mémoire, le CPU et la communication réseau. Il utilise trois mécanismes d'injection différents: 1) fautes mémoires déclenchées après un délai, 2) interruptions (traps) pour générer des fautes CPU non-permanentes, 3) modifications apportées au niveau de quelques instructions du programme lors de la compilation pour émuler des corruptions permanentes d'instruction et de données.

DOCTOR est implémenté sur un système temps réel réparti. L'organisation modulaire de l'outil est particulièrement conçue pour des architectures distribuées. Il est capable de générer des "stimulis" synthétiques et de récupérer des données d'exécution afin d'évaluer la fiabilité d'un système. La faute est injectée avec différentes options (elle peut être permanente, transitoire ou intermittente). La campagne d'injection peut être configurée d'une manière probabilistique, ou en se basant sur des événements passés.

- **Xception**

Xception, développé à l'université de Coimbra au Portugal et décrit dans [J.C. 98], vise des processeurs modernes et complexes. Pour injecter des fautes réalistes et contrôler en détail leur activation et leur impact sur le comportement du système cible, Xception utilise un débogueur avancé et des dispositifs de contrôle d'exécution présents dans la plupart des processeurs modernes d'aujourd'hui. Des fautes sont injectées avec un minimum d'interférence avec

l'application cible (pas de modification au niveau de l'application). Aucune interruption logicielle n'est insérée et il n'est pas nécessaire d'exécuter l'application cible en mode spécial "trace" (l'application est exécutée à pleine vitesse).

Xception fournit un ensemble assez étendu de déclencheurs de fautes, y compris ceux liés à la manipulation des données dans la mémoire. Les fautes injectées par Xception sont de type collage, inversion de bit et court-circuit. Elles peuvent affecter n'importe quel processus fonctionnant sur le système cible (y compris le noyau du système). Il est aussi possible d'injecter des fautes dans des applications pour lesquelles le code source n'est pas disponible.

Des résultats obtenus avec Xception ont par exemple montré sur le processeur PowerPC 601 l'insuffisance des mécanismes de détection d'erreurs. En effet, jusqu'à 73% des fautes injectées produisant des résultats incorrects n'ont pas été détectées dans certaines unités fonctionnelles.

- **EXFI (Exception-based Fault Injector)**

EXFI est un environnement d'injection de fautes développé aux Politecnico de Turin, Italie. Il est dédié à des cartes basées sur un microprocesseur embarqué [A.B. 99]. Le noyau du système EXFI est basé sur le mode d'exception de trace disponible dans la plupart des microprocesseurs. Pendant l'expérience d'injection de fautes, la routine d'exception de trace se charge de calculer le temps d'injection de fautes, d'exécuter l'injection et de déclencher une éventuelle condition de délai. L'outil peut injecter des fautes transitoires de type inversion de bits unique dans l'image de la mémoire du processus (données et code) et dans les registres utilisateur du processeur. L'approche peut être étendue pour inclure différents modèles de fautes, tels que les collages permanents, couplages, inversions de bits avec multiplicité temporelle et spatiale, etc.

Les caractéristiques principales de EXFI sont : un coût réduit (il ne nécessite aucun matériel), une grande vitesse (qui lui permet de considérer un nombre plus élevé de fautes), une grande flexibilité (il supporte différents types de fautes) et une portabilité élevée (il s'adapte facilement pour adresser différents systèmes cible).

- **CEU (Code Emulant un Upset) implémenté par THESIC**

[R.V. 00-a, R.V. 00-b] présentent une méthode pour la prédiction de la sensibilité d'un processeur face aux SEUs qui peuvent survenir lors de l'exécution d'une application. La méthode proposée se base sur l'injection matérielle/logicielle d'erreurs de type SEU sur une carte digitale bâtie autour d'un processeur cible. L'idée est d'utiliser des signaux asynchrones telles que les interruptions pour introduire, lors de l'exécution d'un programme, le changement du contenu d'un bit au sein du processeur. L'exécution de la séquence de traitement de l'interruption appelée CEU (Code Emulant un Upset) provoquera la modification du contenu d'un bit sélectionné aléatoirement parmi les éléments de la zone mémoire accessible au programme (registres, mémoire interne). L'instant de déclenchement de l'interruption est

également choisi aléatoirement. L'implantation de cette technique a été réalisée par l'intermédiaire d'un système THESIC, testeur dédié à la qualification sous radiation de circuits intégrés. Ce système comporte deux cartes digitales (carte mère/carte fille). L'objectif de ces travaux a été de démontrer que le taux d'erreurs d'une application peut être prédit à partir des résultats issus d'essais d'injection de SEUs et des mesures des sensibilités des éléments mémoires du processeur considéré. La confrontation de ces prédictions avec des mesures réalisées à l'aide d'accélérateurs de particules, a permis de montrer la validité de l'approche proposée pour différents types de processeurs.

1.5.2. Avantages et inconvénients

Les approches d'injection de fautes basées sur le logiciel offrent plusieurs possibilités, et **les principaux avantages** sont :

- C'est une technique qui vise des applications et des systèmes d'exploitation, ce qui est difficile à envisager si l'on se focalise sur des techniques d'injection basées sur le matériel.
- Les expériences s'exécutent en temps réel ou presque, ce qui permet de réaliser un grand nombre d'expériences d'injection.
- Cette approche permet d'injecter un grand nombre de modèles de fautes.
- Les techniques employées n'exigent aucun matériel particulier d'où un faible coût d'implémentation.

Par ailleurs, **les principaux inconvénients** sont :

- Il est impossible d'injecter des fautes dans des endroits qui sont inaccessibles au logiciel.
- L'instrumentation du logiciel peut perturber l'exécution de stimuli sur le système cible et peut aussi changer la structure du logiciel original. Une conception prudente de l'environnement d'injection peut minimiser la perturbation des stimuli.
- Dans certains cas, l'observabilité et la contrôlabilité sont limitées. Au mieux, il est possible de corrompre les registres internes du processeur ou des éléments dans la mémoire (qui sont visibles au programmeur).
- L'exécution du programme d'injection de fautes peut affecter l'ordonnancement des processus système, ce qui peut entraîner par exemple des dépassements au niveau des dates limites.

1.6. Injection de fautes basée sur la simulation

1.6.1. Méthodes et outils

La simulation permet d'analyser un système à tous les niveaux, du plus haut (niveau fonctionnel) au plus bas (le comportement élémentaire, transistor, capacités...). La simulation utilise aujourd'hui beaucoup les langages de description de haut niveau tels que VHDL et VERILOG. Ces langages permettent la description de systèmes très complexes aussi bien au niveau comportemental (algorithmes, équations booléennes...) qu'au niveau structurel (niveau portes). De plus, avec ces langages, une description multi niveaux est possible. A un niveau plus élevé, nous pouvons citer les techniques dites de co-simulation. Elles consistent à simuler conjointement la partie matérielle et la partie logicielle du système. Dans un tel environnement, il est possible d'effectuer une co-simulation d'un ensemble de modèles hétérogènes. Par exemple, dans un environnement de co-design dédié à la conception de circuits pour les commandes de processus, la co-simulation consistera à faire communiquer un simulateur de langage HDL, un exécutable provenant d'un programme C et un simulateur du processus électrique (exemple Matlab). A l'autre extrémité, les caractéristiques électriques du circuit pourront être analysées avec une description SPICE. Dans la suite de ce document, nous nous intéresserons plus particulièrement à l'utilisation des langages de haut niveau.

L'injection de fautes basée sur la simulation de description de haut niveau a été proposée dans [J.O. 92, M.R. 92]. Les mécanismes de détection d'erreur d'un processeur RISC 32 bits sont étudiés. L'injection de fautes dans un modèle VHDL au niveau RTL est réalisée en faisant basculer un bit d'un élément interne choisi aléatoirement. La durée d'une faute correspond à un cycle d'horloge et seules les fautes transitoires de type inversion de bit ont été injectées.

Un outil d'injection de fautes (MEFISTO) basé sur la simulation est proposé dans [E.J. 94-a, E.J. 94-b]. L'injection cible des modèles VHDL décrits à différents niveaux d'abstraction. Deux techniques d'injection de fautes sont proposées : la première réalise l'injection en modifiant la description VHDL, tandis que la deuxième utilise des commandes spécifiques à un simulateur dédié. La modification du modèle VHDL consiste à utiliser les saboteurs ou les mutants. Un saboteur est un composant VHDL additionnel qui, lorsqu'il est activé, altère la valeur ou les caractéristiques temporelles d'un ou de plusieurs signaux. Par contre, un mutant correspond à la modification d'un ou de plusieurs composants existants. Lorsque l'injection est inactive, le mutant se comporte exactement comme le composant remplacé et lorsque l'injection est active, il imite un comportement fautif. Dans le cas de l'utilisation des commandes spécifiques à un simulateur, il est possible d'altérer la valeur d'un signal ou d'une variable déclaré dans une description VHDL. Cette technique permet de ne pas modifier le modèle VHDL mais son application dépend des fonctionnalités offertes par le langage de commande du simulateur.

Une technique d'injection de fautes dans des modèles VHDL niveau comportemental est décrite dans [T.D. 96]. Cette méthodologie est développée pour une évaluation de fiabilité. L'injection de fautes cible des instructions au niveau architecture. Le modèle est une description comportementale d'un processeur, des fautes de type "collage à 0, 1 ou x" sont injectées dans différents signaux.

Une approche basée sur la simulation VHDL est décrite dans [A.M. 96]. Le système modélisé en VHDL est une architecture de système temps réel (basée sur le MC68302) utilisée dans des systèmes automatisés pour gérer la lumière du métro. Des fautes sont injectées dans les registres internes du processeur, dans la mémoire externe et dans les bus en utilisant une technique d'altération de code. Le modèle de faute correspond à l'inversion de bit unique; les erreurs provoquées concordent avec les caractéristiques des erreurs produites par l'impact de particules alpha. L'injection est réalisée par des éléments d'injection de fautes additionnels qui sont insérés dans le CPU et les modules relatifs à la mémoire et au bus. Ces parties supplémentaires sont activées par un autre élément additionnel qui est un activateur d'injection de fautes.

Une technique de simulation de fautes orientée défaut est proposée dans [M.S. 99-a, M.S 99-b]. Un simulateur de fautes mixte est présenté. Il implémente des modèles de fautes de type "court-circuit" et "ligne-ouverte" en utilisant le langage Verilog. L'injection de fautes est réalisée exclusivement sur la partie structurelle du circuit. Les deux modèles de fautes Verilog permettent d'injecter des fautes entre les portes logiques ainsi qu'à l'intérieur des portes logiques (en utilisant la bibliothèque des cellules).

À un niveau élevé, un outil de co-design matériel/logiciel permettant de réaliser des expériences d'injection est proposé dans [M.L. 00]. Cette approche est basée sur des simulations de modèles et peut cibler la partie matérielle et logicielle du système. Le modèle de calcul interne de l'environnement de co-design est basé sur des machines d'états finis (CFSM). Deux types d'injection de fautes sont réalisées : injection de fautes comportementale et architecturale. Dans le cas d'une injection de fautes comportementale, les fautes peuvent être injectées en modifiant la séquence des événements émis ou détectés par le CFSM, les valeurs associées aux événements et les valeurs mémorisées dans les variables internes du CFSM. Ceci permet de définir trois types de fautes comportementales : des fautes sur des événements d'entrée, des événements de sortie ou des variables internes. Dans le cas d'événements d'entrée/sortie, trois types de modèles de fautes ont été considérés : suppression d'événement, changement de la valeur d'un événement avant sa détection/émission et détection d'un événement qui n'a pas été émis, ce qui conduit à émettre un événement à un instant erroné. L'injection de fautes dans une variable interne consiste à modifier son contenu. Dans le cas d'une injection de fautes architecturale, le système cible est modélisé comme étant un CPU. Ce dernier inclut une mémoire externe qui contient le programme et les données. Des inversions de bits uniques sont injectées dans la mémoire et dans les registres du CPU.

Des techniques d'injection de fautes basées sur la simulation VHDL au niveau porte, registre et circuit sont présentées dans [J.B. 00, D.G. 00, J.G. 01]. Trois approches importantes sont décrites et comparées: commande de simulateur, mutants et saboteurs, injection de fautes permanentes et transitoires. Dans le cas de l'approche basée sur les commandes de simulateur, quatre modèles de fautes sont employés pour l'injection de fautes transitoires : collage à 0 ou 1, l'indétermination, inversion de bit et délai. Les modèles de fautes permanents sont : collage à 0 ou 1, l'indétermination, inversion de bit et coupure (haute impédance). En cas de saboteurs, les modèles de fautes sont les mêmes, complétés par le court-circuit, et les transistors toujours ouverts ("stuck-open") pour les modèles de fautes permanentes. Les campagnes d'injection ont été exécutées dans un modèle VHDL d'un système micro-ordinateur 16 bits tolérant aux fautes.

Une méthodologie pour injecter des inversions de bits (SEUs) a été proposée dans [F.V. 00]. Les SEUs transitoires sont injectés dans des descriptions VHDL de haut niveau afin d'estimer la fiabilité du circuit. Les sorties primaires du circuit sont observées pour chaque faute transitoire qui affecte le fonctionnement opérationnel du circuit.

La technique proposée dans [B.P. 00] accélère le temps de simulation des campagnes d'injection de fautes. L'accélération des opérations d'injection est atteinte grâce à deux méthodes de base : l'analyse des fautes à injecter afin d'identifier l'effet final de la faute dès que possible et l'exploitation de quelques commandes standard des simulateurs VHDL commerciaux. Le modèle de fautes correspond à une inversion de bits transitoire et unique qui affecte aussi bien des emplacements mémoires contenant des données et du code que les registres utilisateur d'un microcontrôleur. Tout d'abord, une exécution du modèle sans fautes est effectuée, des informations sur le comportement du système et l'état du simulateur sont stockées dans un fichier trace. L'exécution du système avec fautes est comparée périodiquement aux résultats de la simulation sans fautes, et la simulation est arrêtée dès que les effets de la faute sur le système deviennent notoires.

Une technique d'injection de fautes dans des descriptions VHDL en utilisant les mutants est proposée dans [R.L. 00]. La génération des mutants est optimisée de telle manière que des comportements erronés significatifs puissent être forcés pendant les expériences d'injection de fautes. En outre, ces descriptions mutées sont optimisées pour la synthèse matérielle, de sorte que l'on puisse utiliser la simulation et l'émulation à partir de la même description modifiée. Les fautes sont injectées dans la partie contrôle d'une description VHDL synthétisable. Le modèle de fautes correspond aux inversions de bits transitoires. Le modèle d'erreur traité représente à ce niveau de description l'inversion de bits due à des impacts de particules (SEU).

Parmi les outils d'injection de fautes basés sur la simulation et développés dans le domaine de la recherche universitaire, nous pouvons citer les outils suivants:

- **FOCUS**

FOCUS est un environnement de simulation développé à l'université de l'Illinois à Urbana-Champaign (USA) pour valider les circuits intégrés complexes [G.C. 92]. L'injection de fautes

se pratique dans un premier temps au niveau transistor (électrique), ensuite la propagation de la faute est mesurée au niveau porte. L'outil de base de ce simulateur est SPLICE 1 (variante de SPICE). Une méthode statique est utilisée pour accélérer la simulation.

Le modèle de faute utilisé est basé sur un courant parasite altérant la tension sur un nœud. La faute peut être permanente ou temporaire, simple ou multiple. La forme d'onde de l'impulsion de courant est programmable et permet de représenter des parasites fugitifs variés. L'utilisateur contrôle l'emplacement, l'instant d'injection et la durée de la faute.

Ce simulateur a été utilisé pour valider un microprocesseur 16 bits embarqué destiné à des applications en avionique. Il s'agit d'étudier l'impact des fautes transitoires sur un contrôleur de réacteur temps réel afin d'identifier les chemins de propagation de fautes les plus critiques. Le module le plus sensible à la propagation de fautes est celui qui est susceptible de causer le plus d'erreurs externes.

- **MEFISTO**

L'outil MEFISTO développé initialement par le LAAS et l'université de Chalmers [E.J. 94-b], a comme objectif d'injecter des fautes dans des descriptions de circuits. Différents types de fautes et différentes méthodes d'injection sont considérées. La version de l'outil présentée dans [E.J. 94-b] est basée sur la génération de commandes pour un simulateur VHDL pour injecter des fautes en modifiant le statut du simulateur à un temps donné.

D'autres approches d'injection sont implémentées dans une version plus récente développée à Chalmers (MEFISTO-C). Et il est possible d'ajouter des saboteurs dans la description VHDL avec une nouvelle version du LAAS, MEFISTO-L [J.B. 98]. Cependant, l'injection de fautes avec génération des mutants est mentionnée seulement dans le cas des modifications similaires à celles faites dans le test logiciel ou pour des modèles de haut niveau de type "Stuck-then", "Stuck-else" ou défaillance d'opérateur [J.A. 99]. La génération automatisée de tels mutants n'a pas encore été implémentée. De plus, la relation entre les modèles de fautes et le comportement réel d'un circuit en présence de fautes n'est pas claire.

- **VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently)**

VERIFY est développé à l'université de Erlangen-Nurnberg, Allemagne. Il utilise une extension du VHDL pour décrire des fautes en corrélation avec des composants de bibliothèque technologique [V.S. 97]. Cela permet aux fabricants de matériel, ceux qui fournissent des bibliothèques de conception (i.e. porte AND, OR et d'autres descriptions comportementales de composants basiques), d'exprimer leur connaissance du comportement de fautes relatif à chaque composant.

VERIFY utilise aussi le concept des signaux pour pouvoir décrire les fautes en corrélation avec un composant et en même temps avoir une interface pour que le simulateur active la faute

pendant un temps donné. Donc, la description comportementale du composant doit être étendue par les actions corrélées aux fautes.

Par ailleurs, l'injection de fautes multi-tâche utilise des points de contrôle et des comparaisons avec une exécution sans fautes afin d'accélérer la simulation en présence de fautes.

- **Centauri**

Centauri est un compilateur/simulateur commercialisé par FTL Systems pour effectuer des simulations multi-niveaux de circuits intégrés complexes [C.F. 04]. Il se base sur des algorithmes optimisés pouvant être utilisés par des stations multi-processeurs, ce qui permet d'accélérer la simulation. L'outil intègre un module dédié à l'injection de fautes transitoires de type inversion de bit unique et multiple dans un modèle VHDL. La technique permet de ne pas modifier le modèle VHDL mais son application dépend de commandes spécifiques au simulateur.

1.6.2. Avantages et inconvénients

Les approches d'injection de fautes basées sur la simulation offrent plusieurs possibilités, et **les principaux avantages** sont :

- C'est une technique qui permet d'analyser la sensibilité d'un circuit digital complexe très tôt dans le processus de conception.
- L'observabilité et la contrôlabilité ne sont pas limitées.
- La modélisation est multi-niveau : du système au transistor.
- Les techniques employées sont flexibles car une description de circuit peut facilement être modifiée pour injecter des fautes à différents endroits. De plus, une même description peut être utilisée sur différentes plateformes du moment que ces plateformes intègrent le HDL correspondant.
- Elles n'exigent aucun matériel particulier d'où un faible coût d'implémentation.

Par ailleurs, **les principaux inconvénients** sont :

- Les expériences d'injection de fautes nécessitent un temps considérable de simulation.
- Les expériences ne sont pas réalisées sur un circuit réel, ce qui conduit à un manque de précision dans 1) l'estimation de la couverture et de la latence, et 2) l'analyse des chemins de propagation d'erreur.
- La construction de modèles fidèles pose problème, notamment pour la représentation de l'environnement opérationnel.

1.7. Prototypage matériel

1.7.1. Méthodes

L'inconvénient de l'utilisation des simulations est qu'elles nécessitent un temps énorme d'exécution sur CPU, spécialement dans le cas d'un nombre de fautes élevé à injecter dans un circuit complexe. L'émulation permet d'améliorer sensiblement la vitesse de l'injection puisqu'un prototype matériel peut fonctionner de 1.000 à 1.000.000 fois plus rapidement qu'une simulation [R.L. 99]. Le deuxième avantage de l'émulation est la possibilité de connecter le prototype du circuit dans son environnement opérationnel (émulation dans un système externe). Ainsi l'analyse peut être beaucoup plus complète et précise.

Dans le cas de l'émulation, le circuit doit être synthétisé sur des composants programmables. Donc la description initiale du circuit et la description modifiée pour l'injection doivent être synthétisables. Par ailleurs, le nombre de ressources utilisées dans l'émulateur et la fréquence maximale de l'horloge du prototype dépend fortement de la qualité de l'écriture de la description.

L'utilisation de matériel reconfigurable pour réaliser de l'émulation de fautes a eu premièrement pour objectif d'accélérer des simulations de fautes pour du test de production. Ces approches se limitent généralement aux fautes de collage. Cette méthodologie a été introduite premièrement dans [R.W. 94, R.W. 95], et un environnement basé sur les FPGAs a été présenté.

[K.C. 95, K.C. 99] proposent une méthode qui utilise un système d'émulation basé sur les FPGAs pour l'évaluation de taux de couverture. Ils proposent une injection de faute statique et dynamique sans recompilation du circuit. La méthodologie est limitée aux fautes de collage sur les entrées des blocs logiques (CLBs). Les fautes sont injectées en manipulant directement le "bitstream" correspondant aux CLBs affectés (avec des informations obtenues lors de la compilation) et ensuite, en effectuant le chargement du "bitstream" modifié dans le FPGA. Une approche d'injection de fautes dynamique a aussi été proposée et validée à travers des expériences. L'idée de base consiste à dupliquer la fonction dans laquelle la faute sera injectée. Une instance comprendra la fonction originale et l'autre instance correspondra à la même fonction mais avec présence de la faute. À un instant donné, seulement une des deux instances est activée. Dans le cas d'une émulation nominale, la fonction correcte est exécutée et lorsque la faute est injectée dynamiquement, l'instance qui inclut le modèle de faute et le contrôleur d'activation de faute sont activés. Ainsi, un matériel additionnel est nécessaire pour chaque faute à injecter ce qui implique une recompilation du circuit afin d'ajouter ces éléments.

Une méthodologie similaire est proposée dans [J.H. 96]. Elle consiste à injecter des fautes sans recompilation. Le circuit original est converti en circuit instrumenté et implémenté en FPGA. Dans le circuit instrumenté, les fautes sont injectées en décalant le contenu d'une chaîne d'injection de fautes. L'injection se limite uniquement aux fautes de collage.

Une autre approche basée sur l'émulation matérielle est décrite dans [L.B. 96]. Il s'agit d'une méthode d'émulation de fautes en série (SFE) dans laquelle chaque circuit fautif est émulé séparément. Elle est appliquée au circuit niveau porte pour des fautes de type collage et implémentée sur un émulateur matériel commun (Meta System).

Une approche similaire à celles présentées dans [K.C. 95, J.H. 96, K.C. 99] est proposée dans [R.S. 97, R.S. 98-a]. Le circuit est étendu avec des injecteurs de fautes et ensuite implémenté sur le support FPGA. Des injecteurs de fautes sont utilisés pour modéliser des fautes de type "collage à 1" et "collage à 0", ces injecteurs de fautes sont contrôlés par un activateur de fautes. L'idée d'injecter des fautes de type collage est similaire aux approches mentionnées ci-dessus : des portes primitives sont utilisées à cet effet. Cette méthode ne nécessite pas un émulateur logique spécifique. L'expansion du circuit, en utilisant les injecteurs et l'activateur de fautes, a pour résultat un surcoût en ressources FPGA estimé à 3-4 fois plus élevé que le circuit sans logique additionnelle. Une méthode est proposée dans [R.S. 98-b] pour atteindre un meilleur usage des ressources FPGA pour l'émulation de fautes. L'utilisation des ressources est élevée parce que les nœuds du circuit sont assignés aléatoirement aux injecteurs de fautes. Une technique d'optimisation de l'assignement des nœuds a été proposée, qui se base sur l'amélioration du partitionnement et du placement routage du FPGA, ce qui réduit le surcoût de 3-4 fois à 2-3 fois, par rapport au circuit sans logique additionnelle.

Une approche originale a aussi été proposée dans [L.A. 00, L.A. 03], permettant de réaliser des injections sur un prototype sans avoir à instrumenter le circuit cible. Pour cela, les mécanismes de reconfiguration des FPGA à base de SRAM sont exploités et des reconfigurations partielles sont utilisées pour injecter différents modèles de fautes, incluant les SEU. En utilisant cette possibilité, seule une partie du réseau doit être reconfigurée lorsque des modifications sont requises, ce qui conduit à des gains de temps notables lorsque seules quelques différences existent entre deux configurations successives. Chaque injection (ou suppression) de fautes nécessite une reconfiguration partielle du FPGA. L'étude a cependant montré que les architectures de FPGA du commerce sont peu adaptées à ce type d'approche.

Des transformations de circuit pour appliquer des injections d'inversion de bits transitoires uniques dans des circuits émulsés sont proposées dans [P.C. 01-a, P.C. 01-b, P.C. 01-c]. Les analyses de fautes ne nécessitent aucune reconfiguration du FPGA. Le système proposé inclut trois modules principaux (comme la plupart des systèmes d'injection de fautes). Le gestionnaire de liste de fautes se charge d'analyser le système et de générer la liste de fautes à injecter. Pour cela, il a besoin de la description du système niveau portes et des stimuli d'entrée existant. Le deuxième module correspond au gestionnaire d'injection de fautes, il sélectionne une faute, l'injecte dans le système et observe les résultats. L'analyseur de résultat vérifie les données obtenues par le gestionnaire d'injection de fautes. Le gestionnaire de liste de fautes et l'analyseur de résultat ont été implémentés par des processus logiciels. Le gestionnaire d'injection de fautes correspond à une implémentation matérielle/logicielle. La partie logicielle s'exécute sur l'ordinateur hôte et la partie matérielle est localisée sur le support FPGA avec le circuit émulé. La partie logicielle du gestionnaire d'injection de fautes modifie le circuit pour

effectuer les injections. Elle envoie les stimuli d'entrée à la carte FPGA, déclenche l'injection et reçoit de la carte le comportement erroné du système. La partie matérielle du gestionnaire d'injection de fautes est une interface qui intercepte et exécute les commandes issues du gestionnaire. Les éléments additionnels dans le circuit transformé correspondent à des registres qui mémorisent de l'information relative aux bascules qui seront affectées par une inversion de bit, et une logique combinatoire pour injecter la faute. L'injection est contrôlée par un signal d'autorisation. Les expériences ont été exécutées sur un PC classe Pentium standard et sur une carte FPGA de type XCV1000. Les résultats expérimentaux montrent que la transformation proposée du circuit induit un surcoût matériel limité qui diminue en fonction de la taille des circuits considérés.

La technique proposée dans [P.C. 01-a, P.C. 01-b, P.C. 01-c] est étendue dans [P.C. 01-d, P.C. 03] afin de mieux supporter l'injection de fautes dans des systèmes à base de microprocesseur. La partie matérielle du système d'injection de faute (l'interface sur le FPGA) reconnaît les commandes issues du gestionnaire d'injection de fautes, comme décrit ci-dessus. Dans le système [P.C. 01-d, P.C. 03], des commandes spécifiques adaptées aux microprocesseurs peuvent être reconnues par l'interface comme par exemple l'exécution d'une ou de plusieurs instructions du processeur, la lecture ou le changement du contenu d'éléments de mémorisation. Les expériences ont été réalisées sur une carte équipée d'un FPGA Xilinx Virtex V1000 et le modèle du microprocesseur est un Intel 8051. Dans [P.C. 03], des injections de fautes ont été réalisées sur deux modèles de microprocesseur : Intel 8051 et SPARC v8. Les fautes ciblent des unités mémoires, des buses et des registres internes. Les résultats montrent que les expériences d'injection de fautes peuvent être accélérées par cette technique de manière significative.

1.7.2. Avantages et inconvénients

Les approches d'injection de fautes basées sur le prototypage matériel offrent plusieurs possibilités, et **les principaux avantages** sont :

- C'est une technique qui permet d'analyser la sensibilité d'un circuit digital complexe très tôt dans le processus de conception.
- Les expériences d'injection de fautes sont rapides et peuvent être exécutées à fréquence nominale ou proche, ce qui permet d'exécuter un grand nombre d'expériences d'injection de fautes.
- Il est possible de connecter le prototype du circuit dans son environnement opérationnel.
- Les fautes peuvent être injectées à différents endroits.

Par ailleurs, **les principaux inconvénients** sont :

- Certaines contraintes matérielles de l'émulateur peuvent limiter l'observabilité et la contrôlabilité par rapport à une simulation.

- Dans certains cas, le temps nécessaire pour la synthèse et le placement routage sur l'émulateur peut pénaliser les expériences d'injection.
- Le modèle cible doit être synthétisable.

1.8. Injection de fautes hybride

Une approche hybride consiste à mixer différentes techniques d'injection de fautes dans le but de combiner leurs avantages et d'obtenir une analyse plus complète du système. Lors de la conception d'un environnement d'injection de fautes, il existe plusieurs possibilités pour intégrer des systèmes matériel et logiciel. Une des possibilités consiste à combiner un système basé sur un microprocesseur avec du matériel reconfigurable (exemple : environnement basé sur une implémentation FPGA et contrôlé par logiciel).

Un système d'injection de fautes hybride matériel/logiciel est décrit dans [N.K. 95]. L'objectif principal de cette approche est de pallier les limitations des deux méthodes basées sur le matériel et le logiciel. La circuiterie relative à l'injection de fautes matérielle est implémentée en FPGA et le logiciel correspond à une extension de FERRARI. Le module d'injection de faute basé sur le logiciel exécute des séquences d'instructions pour émuler la faute matérielle. Le module d'injection basé sur le matériel applique des signaux logiques sur des lignes sélectionnées et les bus du système cible. Dans les expériences mentionnées, le système cible est le SUN SPARC1 et les fautes sont injectées dans son SBUS pendant que le système exécute différentes applications.

L'injection de fautes basée sur le logiciel est un moyen à moindre coût pour forcer un comportement erroné et modifier ainsi l'état du système cible. Comme précédemment indiqué, l'inconvénient principal de cette technique d'injection est son incapacité d'injecter des fautes à des endroits non accessibles au logiciel. En second lieu, il faut s'assurer que le logiciel additionnel nécessaire pour effectuer les expériences d'injection de fautes, comme les modules d'injection, de contrôle d'expérience, ou le module de collecte et d'analyse de données, ne biaise pas les résultats d'injection de fautes. Dans cette situation, l'injection de fautes basée sur la simulation peut être appliquée offrant une très bonne contrôlabilité des systèmes cibles. Les composants d'un système cible (dans lesquels des fautes seront injectées) sont simulés en utilisant des modèles de ces composants. Ceci permet non seulement de corrompre des données et/ou la synchronisation d'une application, mais rend possible l'injection de n'importe quel type de fautes dans le système cible.

La combinaison entre le logiciel et la simulation est une méthode décrite dans [J.G. 95] et développée à l'université de technologie de Chalmers, Suède. Cette technique hybride d'injection de fautes, également connue sous le nom d'injection de fautes en mode mixte, permet de tirer profit des avantages des deux méthodes d'injection. Le système cible réel peut être exécuté à pleine vitesse excepté pendant l'injection d'une faute où un simulateur fournit un accès détaillé au système cible. En effet, lors de l'activation de l'injecteur de fautes, le

fonctionnement nominal du système cible est interrompu et l'état du système est transféré au simulateur. C'est à ce moment que les données peuvent être altérées et/ou la synchronisation du système influencée par le logiciel (c.-à-d. par l'injecteur de fautes logiciel qui simule la faute). À la fin de la phase de simulation du système, son état modifié est réécrit et le système continue normalement l'exécution des opérations. Une comparaison entre une exécution de référence (sans fautes) et une exécution avec injection de fautes est nécessaire pour déterminer les états d'erreur.

Etant donné que pratiquement tous les systèmes informatiques supportent le traitement multitâche et/ou le point de contrôle "checkpoint", ils peuvent sauvegarder et reconstituer le contenu du processeur et du coprocesseur. Dans la plupart des cas, il est possible d'obtenir un état cohérent de tout le système, qui peut être transféré à un simulateur. Il n'est pas nécessaire de transférer au simulateur le contenu de tous les registres et des cellules mémoires mais seulement les informations qui seront utilisées par le simulateur. Puisque peu de registres et de cellules mémoires sont modifiés dans la phase courte d'injection de fautes, seulement une petite partie de l'état du système est copiée.

Pour réduire le nombre d'injections de fautes, la technique est combinée avec l'injection de fautes basée sur le profil opérationnel. Cela consiste à injecter des fautes seulement dans des éléments (par exemple registres) qui contiennent les données critiques.

LIVE, développé à Ansaldo-Cris en Italie et décrit dans [A.A. 97], intègre des techniques d'injection de fautes et de test logiciel pour réaliser une analyse précise et non-intrusive d'un prototype de système. Il utilise deux approches d'injection différentes : le **forçage niveau broche** et la **génération d'interruption** pour activer les procédures d'injection de fautes. L'outil a été utilisé pour évaluer un système de contrôle ferroviaire informatisé basé sur un CPU de type Motorola 68040. Les résultats issus des expériences d'injection de fautes effectués sur la carte Motorola MVME162 (qui exécute un analyseur évaluant des expressions algébriques) montrent clairement l'impact du domaine d'entrée sur la fiabilité.

Une technique d'injection de fautes hybride logicielle/FPGA est aussi proposée dans [L.B. 02]. La plateforme d'injection de fautes a été développée dans un environnement industriel réel, et le modèle de faute appliqué est l'inversion de bit unique. Quelques fonctions de contrôle sont implémentées en FPGA, le reste du système est implémenté en logiciel. Le but de la campagne d'injection de fautes est d'obtenir une classification des éventuelles fautes qui produisent une différence au niveau des ports de sortie du circuit pendant l'expérimentation ou bien au niveau des éléments mémoires à la fin de l'expérimentation.

1.9. Comparaison entre les techniques d'injection de fautes

Les principaux avantages et les inconvénients des techniques d'injection de fautes vues précédemment sont résumés dans le Tableau 1-1.

Tableau 1-1: Résumé des avantages et inconvénients des différentes techniques d'injection de fautes

Approche	Avantages	Inconvénients
Injection physique sans contact (bombardement par des particules, au laser)	- Systèmes et fautes réels - Finesse du test (niveau transistor) - Accès à toutes les zones du circuit	- Validation tardive - Installations de test lourdes et coûteuses - Résultats non reproductibles (particules)
Injection physique avec contact (niveau broche)	- Système et fautes réels	- Validation tardive - Accessibilité limitée - Modèles de fautes insuffisants - Problèmes d'interfaçage
Logiciel	- Application/Système d'exploitation ciblé - Exécution en temps réel - Faible coût d'implémentation	- Risque de perturber la structure du logiciel original - Observabilité/contrôlabilité limitée
Simulation	- Validation très tôt dans le développement - Tous niveaux de description possibles : du système au transistor - Faible coût d'implémentation	- La construction de modèles fidèles pose problème - Nécessite un temps considérable de simulation - Simulation logicielle (éloignée de l'environnement opérationnel)
Prototypage matériel	- Validation précoce dans le développement - Plus rapide que la simulation - Plus flexible que les injections physiques	- Exige des modèles synthétisables - Moins rapide que les injections physiques - Moins flexible que la simulation

Face à la complexité croissante des systèmes (comportant des ASIC, des microprocesseurs...), l'analyse pratique devient de plus en plus difficile. C'est ce qui nous donne à penser que la simulation ou le prototypage prendra peu à peu le pas sur l'injection physique dans le processus de validation. D'autant plus que certains outils sont disponibles. Toutefois, la construction de modèles fidèles pose problème. Par ailleurs, l'injection physique sur le système réel reste actuellement partiellement incontournable pour les étapes de qualification dans les domaines d'application critiques.

L'injection de fautes par logiciel sur PC ou station est assez aisée à mettre en œuvre et très souple d'utilisation. De plus, de très nombreux modèles de fautes peuvent être implantés grâce à la puissance et la capacité de stockage des PC et/ou des stations de travail actuels. Elle reste cependant essentiellement limitée à des systèmes à base de microprocesseur. Dans ce contexte, l'inconvénient des approches qui se basent sur le prototypage matériel est lié à l'exportation du

logiciel du système cible sur un microprocesseur hôte différent de ce système. Des problèmes d'interprétation, de compatibilité, d'exécution, de temps réels... se posent alors. Cet inconvénient n'existe pas dans la technique d'injection de fautes basée sur le logiciel. Effectivement, le logiciel instrumenté est celui d'origine, modifié et exécuté sur le système réel lui-même. Comme dans la technique précédente, de nombreux modèles de fautes sont possibles.

1.10. Conclusion

La conception de circuits VLSI sûrs exige des techniques et des outils pour une évaluation prédictive de l'efficacité des mécanismes de durcissement contre les effets des fautes envisagées dans l'application. En particulier, les concepteurs sont intéressés par des outils capables de réaliser des campagnes d'injection de fautes avant que le circuit ne soit fabriqué. Dans ce cadre, les techniques basées sur le matériel ne peuvent pas être utilisées. Les techniques basées sur le logiciel ont le même inconvénient et ne permettent généralement pas d'injecter des fautes dans toutes les zones d'un circuit.

Les techniques d'injection de fautes basées sur la modification d'une description de haut niveau offrent des perspectives prometteuses par rapport aux problèmes cités ci-dessus. Elles permettent, lorsque l'on dispose uniquement du modèle d'un système, d'obtenir une évaluation "très tôt" de la sûreté de fonctionnement de ce système. Par ailleurs, elles sont très flexibles en terme de modèle de faute puisque tous les modèles de fautes peuvent potentiellement être supportés et les fautes peuvent être injectées dans n'importe quel module du système. Leur principale limitation, à savoir le temps de simulation requis, peut être réduite en utilisant une approche à base de prototypage.

La suite de ce manuscrit concernera donc la proposition d'un environnement d'analyse basé sur des simulations de descriptions de haut niveau (VHDL RTL), le processus d'injection de fautes étant toutefois entièrement compatible avec la réalisation d'expériences utilisant un prototype sur FPGA.

Chapitre 2.

Flot d'analyse proposé

2.1. Objectifs du travail

L'injection de fautes est l'une des principales approches pour l'analyse des fonctionnements aberrants pouvant résulter d'une faute transitoire ou permanente. Comme nous l'avons déjà mentionné, il est important de pouvoir réaliser cette analyse le plus tôt possible dans le processus de conception. L'identification tardive de comportements inacceptables a pour conséquence le report des délais de mise en production et l'augmentation du coût de développement. Il y a quelques années, des méthodes d'injection ont été proposées pour intervenir dans des modèles HDL en cours de conception. Modifier des descriptions de haut niveau (RTL), analyser l'effet de l'injection de fautes et valider des caractéristiques de sûreté très tôt dans le flot de conception permet justement d'éviter des itérations très coûteuses et très lourdes dans le flot de conception.

Pour cela, l'approche que nous proposons se base sur l'injection de fautes dans des descriptions de haut niveau (description VHDL au niveau RTL) et sur la génération d'un modèle représentant les comportements consécutifs à la faute. Ce modèle comportemental est dérivé de résultats de simulations ou d'exécutions sur un prototype matériel. Une telle analyse est indispensable pour décider de la stratégie de tolérance la mieux adaptée pour chaque circuit, et sa complexité impose de disposer d'outils automatisant l'essentiel du travail. Le travail réalisé pendant cette thèse a donc consisté à concevoir et à implémenter un environnement de CAO performant permettant d'analyser, très tôt dans le flot de conception, les effets potentiels d'une faute sur une application exécutée par un circuit intégré numérique.

2.2. Flot global

L'un des centres d'intérêt principal de nos analyses est la propagation des erreurs provenant du phénomène SEU. Les types de fautes visés dans le cadre de la thèse sont donc les inversions de bits uniques et multiples (MBF) dans des éléments mémoires et les transitions erronées dans des registres d'états. Ils sont décrits plus en détail dans la section 1.2.1.6.

2.2.1. Vue d'ensemble

L'environnement de CAO développé a pour but de générer automatiquement des modèles fonctionnels de haut niveau aidant le concepteur à identifier et à quantifier les modes de défaillance d'un circuit (ou d'un bloc réutilisable, ou IP). Le résultat d'analyse doit montrer les comportements erronés et les propagations d'erreur réellement observée pour un ensemble donné de fautes et pour une application donnée. L'ensemble de solutions alternatives que nous avons considéré est illustré dans la Figure 2-1.

La description initiale du circuit peut être instrumentée ou utilisée sans aucune modification. Dans notre cas, l'instrumentation consiste à transformer le code VHDL avant synthèse et peut utiliser des saboteurs pour des modèles simples de faute, ou des mutants pour injecter des modèles de faute plus complexes. Cette alternative est discutée dans la section 2.2.3 et plus de détails sont donnés dans le chapitre 3 au sujet de la génération des mutants.

La campagne d'injection peut alors être exécutée en utilisant la simulation ou l'émulation. Notre but est d'être compatible avec n'importe quel flot de conception en utilisant n'importe quel simulateur commercial. Par conséquent, les approches d'injection utilisant des primitives de simulation ou un simulateur dédié n'ont pas été implémentées. Ceci implique que dans le cas d'une campagne basée sur la simulation, la description du circuit doit être modifiée.

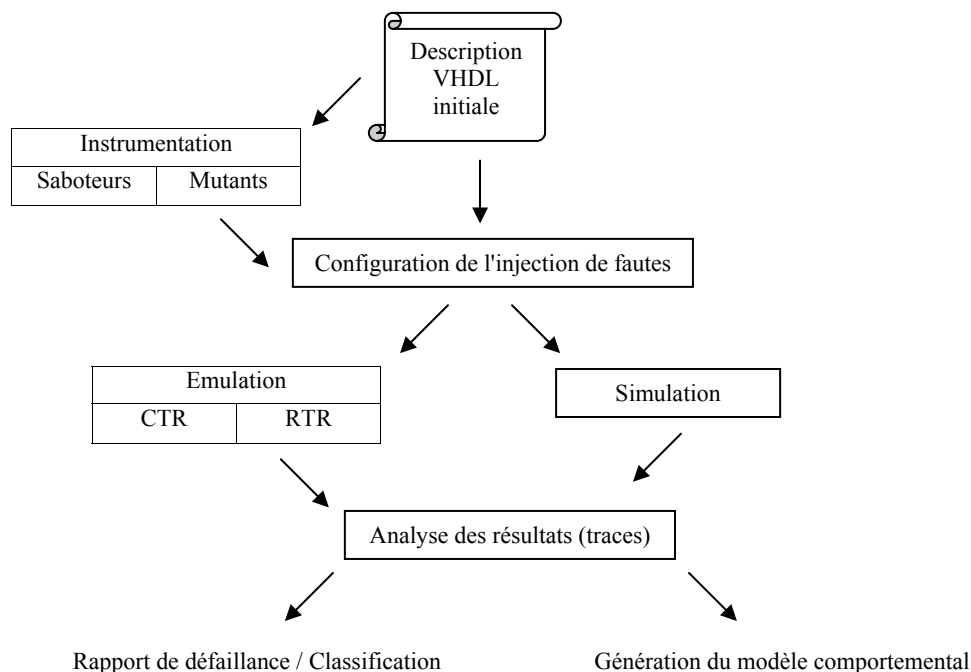


Figure 2-1: Vue globale du flot d'analyse.

Dans le cas d'une campagne basée sur l'émulation, nous considérons deux approches. La première, fondée sur une reconfiguration statique (*Compile-Time Reconfiguration*, CTR), sera utilisée lorsque la description du circuit est instrumentée. Dans ce cas, l'architecture du circuit est chargée sur l'émulateur et la configuration matérielle reste statique jusqu'à la fin de

l'exécution. La deuxième approche, basée sur la reconfiguration dynamique (*Run-Time Reconfiguration*, RTR), permet de réaliser des injections sans modifier la description initiale du circuit. Les fautes sont injectées directement dans le circuit pendant l'exécution sur l'émulateur. Cette approche a fait l'objet de la thèse de Lorinc Antoni [L.A. 03-a] et ne sera pas davantage abordée dans ce document.

Dans le cadre de cette thèse, nous nous sommes focalisé sur certaines approches utilisant des modifications du circuit. Le flot global d'analyse est résumé dans la Figure 2-2. Pendant la définition de la campagne, le concepteur fournit les informations nécessaires pour l'injection de fautes et l'analyse des résultats.

A partir d'une description VHDL (synthétisable), le concepteur doit définir le(s) modèle(s) de fautes, les cibles d'injection et les noeuds à observer. Les expériences d'injection sont réalisées soit par simulation soit par émulation matérielle.

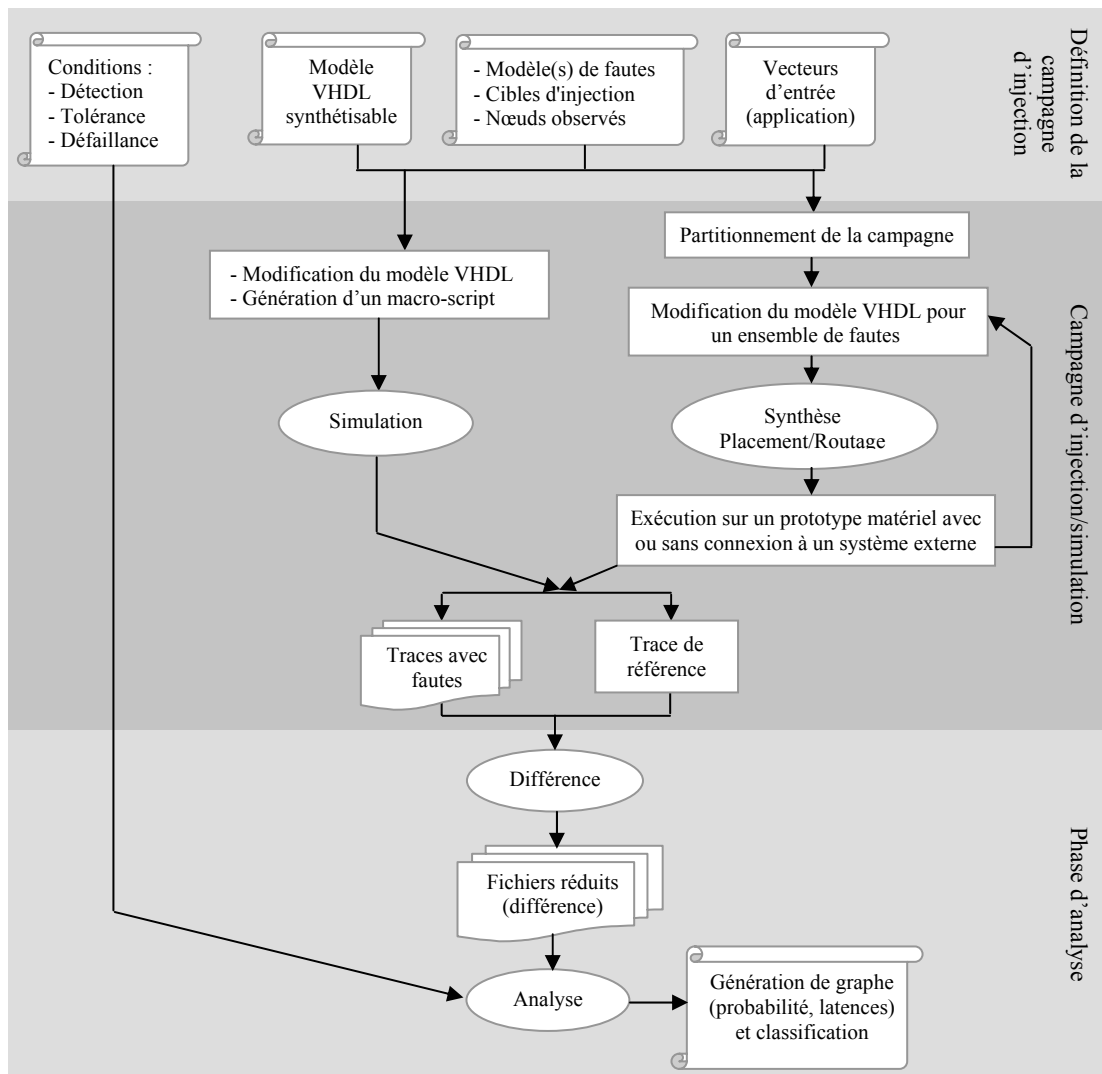


Figure 2-2: Flot global d'analyse pour une description modifiée de circuit.

Dans le cas de la simulation, la description VHDL initiale est modifiée de façon à rendre possible l'injection de toutes les fautes choisies aux endroits sélectionnés. Un "macro-script" contrôle le processus d'injection en activant les fautes aux instants spécifiés. Ce script se base sur des distributions spatiale et temporelle des fautes (déterministes ou aléatoires) spécifiées par l'utilisateur. Les traces de simulation (ou les résultats obtenus par émulation matérielle) sont sauvegardées dans des fichiers.

Dans le cas de l'émulation, les contraintes matérielles de l'émulateur peuvent exiger plus de modifications et exclure l'exécution d'une seule description modifiée. Les raisons seront détaillées dans la section 2.2.2. Dans ce cas, la campagne d'injection doit être divisée en plusieurs sous-campagnes. Pour chacune d'entre elles, des modifications spécifiques seront apportées à la description initiale. Chaque description modifiée doit être synthétisée, placée et routée sur l'émulateur. Selon les objectifs de la campagne d'injection, les exécutions peuvent impliquer l'émulation du circuit dans le système opérationnel.

Les résultats issus de la campagne d'injection correspondent à des traces sauvegardées dans des fichiers. Ils montrent l'évolution des signaux observés et sont indépendants de l'approche utilisée pour exécuter les expériences (simulation ou émulation, avec ou sans modification) puisque seulement une analyse fonctionnelle du circuit est réalisée (aucune donnée temporelle n'est sauvegardée, seulement l'état des signaux échantillonnés à chaque cycle d'horloge). Ces traces sont alors utilisées pour effectuer une analyse comportementale du circuit en présence de fautes. Cela consiste à comparer le comportement de référence du circuit avec le comportement obtenu en présence de chaque faute de l'ensemble de fautes prédéterminé. Cela permet de détecter l'apparition d'événements redoutés et de fournir au concepteur des résultats synthétiques (qualitatifs et quantitatifs). Les conditions significatives associées aux événements critiques (défaillance, détection...) sont spécifiées par le concepteur. Les résultats quantitatifs incluent la probabilité de transition vers un état erroné donné et la latence observée pour passer d'un état à un autre.

2.2.2. Simulation versus émulation

Les avantages que l'on peut attendre d'une émulation sont d'améliorer la vitesse de la campagne d'injection [R.L. 99] et/ou d'obtenir une analyse plus détaillée, en connectant le prototype du circuit dans son environnement opérationnel (émulation dans un système externe). Cependant, l'émulation possède également quelques limitations et inconvénients qui doivent être pris en considération à différentes étapes du flot d'analyse. Les contraintes additionnelles principales, comparées à la simulation, sont :

- le besoin d'une description synthétisable de circuit. Ceci signifie que la description initiale doit non seulement être conforme au langage VHDL (si VHDL est le langage de description choisi), mais doit également satisfaire les contraintes de synthèse. En outre, le nombre de ressources utilisées sur l'émulateur et la fréquence maximum de l'horloge du prototype dépend fortement du style d'écriture de la description. Le code VHDL initial écrit par le

concepteur devrait donc être optimisé pour la synthèse afin d'éviter l'utilisation d'un émulateur coûteux et pour augmenter aussi la fréquence de l'horloge, de ce fait réduisant le temps d'exécution total pendant la campagne d'injection.

- la nécessité d'optimisations dans le processus d'instrumentation (quand l'approche RTR n'est pas utilisée). Dans la plupart des cas, aucun dispositif n'est disponible pour injecter une faute dans un prototype. Par conséquent, des éléments logiques doivent être rajoutés avec des signaux de commande externes. Les modifications sont souvent semblables à celles utilisées dans les approches basées sur la simulation, mais la contrainte de synthèse doit également être prise en compte pour la description instrumentée. On montrera dans la section 3.2 que plusieurs types de modifications, permettant au concepteur d'injecter le même ensemble de fautes ou d'erreurs, peuvent avoir des conséquences très différentes sur les caractéristiques du prototype. En utilisant l'émulation, les modifications doivent être soigneusement optimisées pour améliorer la qualité du prototype à implémenter.

Par ailleurs, en utilisant l'émulation, des sorties primaires additionnelles sont souvent exigées pour observer quelques signaux internes, particulièrement dans le cas d'une analyse détaillée des chemins de propagation d'erreur. Le nombre d'entrées et de sorties supplémentaires ajoutées à la description initiale a un impact sur l'implémentation du prototype car le nombre d'E/S détermine souvent la taille de l'émulateur utilisé (et donc son coût) [R.L. 01]. Lorsque le nombre d'E/S total du circuit instrumenté est supérieur au nombre d'E/S sur le plus grand émulateur disponible, il est nécessaire de diviser la campagne en sous-campagnes, et de générer pour chaque sous-campagne une description instrumentée correspondant à un sous-ensemble donné des fautes ciblées (pour réduire le nombre d'entrées de commande supplémentaires) et/ou à un sous-ensemble donné de signaux observés (pour réduire le nombre de sorties primaires supplémentaires). Cette partition en sous-campagnes augmente sensiblement le temps de préparation des expériences (plusieurs synthèses, placements et routages) et également le temps d'exécution. Il est donc crucial d'optimiser la description instrumentée de telle sorte que le nombre de signaux externes additionnels soit réduit au minimum.

Quelques autres contraintes ou limitations doivent aussi être prises en considération lorsqu'on utilise l'émulation. Par exemple, il n'est pas possible d'analyser les signaux à trois états ou à valeurs indéterminées; les signaux seront échantillonnés comme étant des uns ou des zéros, bien qu'ils puissent être considérés à l'état Z ou X par un simulateur. En outre, les modèles de faute et les cibles d'injection de faute doivent être soigneusement choisies pour être significatives. Puisque la structure bas niveau du prototype du circuit n'est pas identique à la structure du circuit final, les cibles d'injection (et les signaux observés) doivent inclure uniquement des noeuds qui existeront indépendamment de la technologie utilisée. Le plus souvent, des erreurs sont injectées plutôt que des fautes. Des fautes structurelles peuvent également être considérées, mais seulement sur les interconnexions entre les blocs et si la synthèse du circuit final est faite hiérarchiquement. Dans les autres cas, le noeud sur lequel la

faute est injectée peut par la suite disparaître dans la structure finale et l'analyse serait donc sans signification.

L'avantage de la simulation est la possibilité d'effectuer des injections de fautes sur des descriptions de haut niveau qui ne sont pas (entièrement) synthétisable. L'émulation est au contraire mieux adaptée dans le cas d'un circuit qui utilise des IPs complexes dont le modèle détaillé n'est pas disponible. Il est plus facile de connecter un émulateur à un modèle matériel de l'IP (par exemple un processeur encapsulé correspondant à l'IP choisie).

En conclusion, l'efficacité de l'émulation en termes de vitesse n'est pas systématique; elle dépend de plusieurs paramètres, et en particulier :

- de l'optimisation de l'instrumentation, comme discuté ci-dessus,
- de la configuration de l'émulateur utilisé, et particulièrement de la largeur de bande passante de la liaison entre la logique programmable et l'ordinateur principal utilisé pour stocker les résultats [R.L. 01],
- du type d'expérience effectué (classification des fautes ou génération d'un modèle comportemental, ce qui sera détaillé dans la section 2.3) puisque la quantité de données à transférer pendant les expériences est beaucoup plus importante pour une analyse des chemins de propagation, un analyseur de réponse intégré ne pouvant être utilisé que pour la classification.

Les contraintes directement liées à l'émulateur (notamment, le partitionnement de la campagne) ne seront pas considérées dans cette thèse. Toutefois, les méthodes d'instrumentation développées viseront à permettre une émulation efficace.

2.2.3. Saboteurs versus mutants

Deux approches différentes peuvent être utilisées pour modifier la description initiale du circuit. La première consiste à insérer, entre les blocs existants, quelques blocs supplémentaires pour exécuter l'injection de fautes. De tels blocs sont appelés saboteurs [E.J. 94-b]. Ce genre de modification est facile à réaliser et nécessite surtout la modification de certaines interconnexions dans la description initiale. Le saboteur peut injecter certains types de fautes dans les interconnexions modifiées. Cependant, il est presque impossible d'injecter des erreurs modélisées à haut niveau (comportementales) telles que des transitions erronées. En général, il est aussi impossible d'injecter des fautes (exemple : des inversions de bits) dans certains éléments internes aux blocs. Pour de telles fautes, la modification doit s'effectuer directement dans les blocs de la description initiale, ce qui est plus difficile à réaliser mais beaucoup plus puissant. Dans ce cas, la description modifiée du bloc est appelée un **mutant**.

Dans ce qui va suivre, nous utiliserons ce terme pour indiquer les modifications dans la description fonctionnelle d'un bloc, par opposition avec les modifications structurelles (ajout ou remplacement de bloc, modification d'interconnexions).

L'insertion de saboteurs dans une description VHDL est assez facile à réaliser et a déjà été présentée par différents auteurs (par exemple [E.J. 94-b] ou [J.G. 01]). Ce point ne sera donc pas détaillé ici mais nous détaillerons la génération de mutants dans les chapitres 3 et 4.

2.3. Résultats d'analyse

2.3.1. Classification

La plupart des outils décrits dans la littérature visent à classifier les fautes, c'est à dire identifier les ensembles de fautes associés aux différents modes de défaillance ou de détection pouvant être activés. Ceci correspond à la génération d'un modèle simple (Figure 2-3) représentant le comportement en présence de fautes. Une telle classification permet surtout à un concepteur d'évaluer l'efficacité des mécanismes de détection de fautes et le taux de défaillance critique du circuit. L'outil développé réalise ce type d'analyse classique. Cependant, un modèle comportemental peut également être produit, montrant les chemins internes de propagation d'erreur avant occurrence d'une défaillance ou d'une détection, et fournissant des informations plus détaillées au concepteur. C'est l'une des principales différences par rapport aux autres environnements décrits dans la littérature. Ce second type d'analyse est détaillé dans la section suivante.

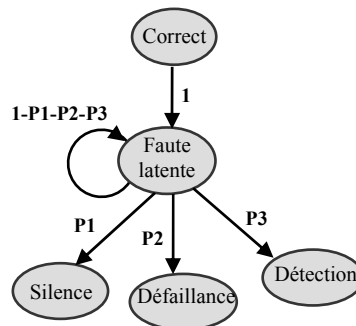


Figure 2-3: Exemple de modèle obtenu en classifiant l'effet des fautes

2.3.2. Génération d'un modèle comportemental de propagation d'erreur

La Figure 2-4 illustre la structure générale du modèle comportemental visé par notre environnement d'analyse. Il correspond à un modèle fonctionnel représentant le comportement du circuit en présence de fautes. Dans notre approche, un tel modèle est dérivé automatiquement des traces de simulation ou d'émulation obtenues pendant les expériences d'injection [R.L. 99]. Le résultat est un graphe où les sommets correspondent aux seuls états réellement atteints par le circuit pendant les expériences. À chaque état est associée une liste donnée de signaux incorrects parmi ceux observés pendant les expériences (la liste de signaux observés est spécifiée par l'utilisateur). Chaque état est également étiqueté avec un type fonctionnel :

- **Correct** : avant l'introduction de la faute,

- **Latent** : la faute ou l'erreur injectée n'a pas encore provoquée une configuration incorrecte des signaux observés,
- **Erreur** : une faute a été activée et au moins un signal observé est incorrect,
- **Sans erreur** : une faute a été activée mais il n'y a pas d'erreur dans les signaux observés,
- **Dégradé** : le circuit est opérationnel mais en mode dégradé,
- **Détection** : des mécanismes de détection de faute/erreur ont été déclenchés,
- **Tolérance** : la faute/erreur activée est tolérée par des mécanismes internes,
- **Défaillance** : une erreur inacceptable se produit.
- **Crash** : un évènement inacceptable pour le simulateur provoque l'arrêt d'une simulation au niveau RTL.

Les états correspondent donc aux différentes configurations d'erreurs activées pendant les expériences. Cette analyse peut être vue comme une extension de la classification, recherchant non seulement l'état final du circuit (défaillance, détection, ...), mais aussi les états internes intermédiaires qui ont mené à cet état final. Une différence importante entre les deux types d'analyses est que l'étude des propagations d'erreur exige de comparer avec précision (à chaque cycle d'horloge) la valeur de chaque signal observé avec la valeur attendue, tandis que la classification de fautes nécessite seulement de détecter l'occurrence d'un nombre limité de configurations d'erreurs à n'importe quel moment de la campagne d'injection.

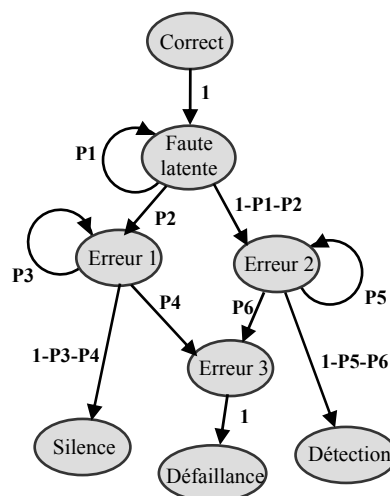


Figure 2-4: Exemple de modèle obtenu en analysant les chemins de propagation des erreurs.

L'analyse des résultats d'une expérience donnée peut s'interrompre dans certaines conditions définissant des états terminaux (généralement liés à une détection ou à une défaillance). Les conditions pour les modes dégradé, détection, tolérance ou défaillance sont spécifiées par l'utilisateur et sont optionnelles. Par ailleurs, les conditions peuvent être très précises, incluant l'état exact de certains signaux ou des propriétés d'évolution par rapport au temps. De cette façon, il est possible de prendre en compte le fait que différents modes de défaillance n'ont pas le même impact sur l'application. Par exemple, les résultats publiés

supposent souvent que n'importe quelle erreur propagée aux sorties primaires du circuit correspond à un état de défaillance. Ceci n'est pas vrai dans la pratique. La criticité dépend par exemple de l'instant d'occurrence (par exemple, les données erronées sur un bus de sortie ont une conséquence seulement si le signal de commande indiquant la validité de ces données est actif autrement les données ne sont pas lues). Elle dépend également des conséquences exactes sur l'application (par exemple, une erreur au niveau du bit de poids faible d'un bus pourrait être négligée ou juste considérée comme une erreur, alors qu'une erreur au niveau du bit de poids fort sur le même bus pourrait être considérée comme une défaillance).

L'état "crash" correspond à un état terminal particulier. Il peut être ajouté dans un graphe généré à partir de résultats de simulation au niveau RTL, car à ce niveau, une simulation peut s'arrêter au milieu de l'exécution, en raison de l'occurrence d'un événement que le simulateur ne peut pas résoudre. Par exemple, une faute peut conduire une variable entière à prendre une valeur hors de son intervalle. D'autres aspects des limitations de la simulation comportementale sont présentés dans [A.A. 05, A.A. 04-b]. Ils sont liés à certains manques d'informations, par rapport aux simulations au niveau portes : notamment, manque d'informations temporelles (en général, un modèle sans retard est employé) et méconnaissance des valeurs finales qui seront prises par les signaux phi-Booléens.

Les transitions observées entre les différents états sont enregistrées. La propagation d'erreur commence à partir de l'état où la faute est injectée (état "Latent") et se termine souvent par un état correspondant à un mode de défaillance ou de détection, comme illustré dans la Figure 2-4. Dans certains cas, selon la structure du circuit et l'application choisie, la propagation peut également s'arrêter dans n'importe quel autre type d'état pour une expérience d'injection donnée.

Une probabilité est associée à chaque transition. Elle est calculée en fonction de l'activité du système pendant les expériences. Le nombre qui est indiqué sur l'arc entre l'état S_i et l'état S_j est obtenu comme le pourcentage des cas où, pendant l'ensemble de la campagne d'injection, l'état S_j a été enregistré juste après l'état S_i (probabilité d'activer l'état S_j lorsqu'on se trouve dans l'état S_i , conditionnée par l'application exécutée et l'ensemble des fautes injectées). La somme des probabilités sur les arcs avec le même état initial est toujours égale à un (une boucle sur un état donné est possible) et le modèle peut être représenté avec une matrice stochastique, comme un modèle de Markov à événements discrets.

Le modèle visé fournit des informations sur le comportement du système selon trois hypothèses :

- L'occurrence d'un type donné de fautes ou d'erreurs sur un ensemble donné de signaux cibles.
- Une liste de signaux et éventuellement d'événements significatifs à observer.
- Un programme de test donné, devant être représentatif du déroulement d'une application.

En plus de la modélisation comportementale, des latences peuvent également être indiquées, précisant le nombre de cycles (minimum, maximum ou moyen) passé dans chaque état avant que la configuration d'erreur n'évolue.

Le modèle produit à la fin de l'analyse montre la propagation des erreurs dans le circuit et permet au concepteur d'identifier les modes de défaillance inacceptables. Il permet aussi de l'aider à définir les meilleurs mécanismes à implémenter dans des endroits précis du circuit, afin d'éviter les défaillances critiques ou de réduire la probabilité d'une propagation d'erreur donnée. Finalement, il peut faire partie de la documentation stockée ou même fournie avec le circuit. De tels modèles peuvent, par exemple, donner des informations pertinentes sur les comportements erronés qui devraient être pris en considération lorsqu'il s'agit d'inclure un composant réutilisable (bloc d'IP) dans un circuit. Fournir ce type d'information avec une description d'IP deviendra crucial avec l'utilisation des systèmes-sur-puce dans des applications ayant une criticité moyenne ou élevée.

Il est à souligner que le concepteur n'a pas à préciser ce que seront les chemins de propagation. Il a juste à définir les signaux significatifs à observer (par exemple, des Entrées/Sorties critiques pour certains blocs ou quelques variables associées à des registres) et les conditions de détection, de tolérance ou de défaillance. Les états réels du modèle sont dérivés des résultats d'expérience et si quelques conditions définies par le concepteur ne se produisent jamais, aucun état correspondant ne sera inséré dans le modèle produit. C'est l'une des différences principales avec les approches précédentes [E.J. 94-b, S.S. 97]. Cela a pour conséquence de simplifier le travail du concepteur en basant son travail sur des modèles précis mais concis du comportement du circuit. Des exemples significatifs de génération de modèles seront donnés dans le chapitre 5.

2.3.3. Une approche hiérarchique

Dans l'approche proposée, chaque bloc dans un circuit peut être étudié indépendamment des autres. Le modèle comportemental obtenu pour un bloc donné peut être utilisé pour déterminer ou préciser le modèle d'erreur appliqué à un niveau hiérarchique plus élevé. A terme, il est envisageable d'utiliser une approche "diviser et conquérir" pour faire face à la complexité des circuits et évaluer en plusieurs étapes les chemins possibles de propagation d'erreur. Les modes de défaillance déterminés au niveau circuit peuvent aussi être utilisés comme modèle d'erreur pour déterminer les propagations au niveau équipement (carte). Les travaux futurs incluent donc la génération de modèles hiérarchiques, de telle sorte que, par exemple, le modèle obtenu pour un bloc d'IP puisse être directement utilisé comme un sous-modèle pour le circuit qui intègre cet IP.

2.3.4. Portabilité et raffinement bas niveau du modèle généré

Les probabilités de transition calculées pour un modèle comportemental obtenu d'une description initiale à haut niveau ne peuvent pas être représentatives de l'effet des fautes

structurelles à l'intérieur des blocs (par exemple : fautes de collage dans la logique combinatoire), puisque la structure finale de ces blocs n'est pas encore connue. En outre, la probabilité de mémorisation d'un aléa dans les parties combinatoires du circuit peut être évaluée seulement à partir d'une netlist niveau portes, une fois que les temps de propagation sont connus de manière exacte. Si de telles fautes de bas niveau doivent être considérées, il est nécessaire de raffiner l'analyse une fois que la structure finale du circuit est disponible, de sorte que des probabilités plus précises peuvent être calculées pour les modes de défaillance.

La simulation de fautes classique peut être utilisée pour évaluer la probabilité que des fautes survenant dans un bloc combinatoire aboutissent à un type d'erreur considéré lors de l'analyse sur la description de haut niveau. Il est alors possible de réinjecter cette probabilité dans le modèle comportemental généré en multipliant la probabilité d'une propagation donnée (conditionnée par l'erreur considérée à haut niveau) par la probabilité que la faute structurelle induise le type d'erreur initialement considérée. Puisque la simulation de fautes peut être effectuée indépendamment pour chaque bloc, le temps global de simulation est sensiblement réduit par rapport à une approche qui nécessite une simulation de fautes du circuit global. Evidemment, un tel raffinement du modèle n'est possible que si les erreurs, injectées tôt dans le flot de conception, correspondent à des effets des fautes considérées comme significatives dans les analyses ultérieures.

Puisque la modélisation de haut niveau est alors indépendante de la quantification bas niveau, la même description qualitative des modes de défaillance potentiels peut être utilisée pour différents modèles de fautes bas niveau et diverses structures niveau portes. En cas de changements de technologie, le modèle haut niveau est donc réutilisable et la quantification pour la nouvelle technologie ou le nouveau modèle de faute peut être obtenue beaucoup plus rapidement qu'avec des approches classiques de simulation de fautes.

2.4. Conclusion

Le besoin de modéliser le comportement des circuits complexes consécutif aux fautes ne cesse d'augmenter. Le flot d'analyse actuellement automatisé dans notre environnement vise à générer un modèle détaillé montrant les chemins de propagation d'erreur dans le circuit. Il offre plusieurs alternatives pour faire face aux différentes contraintes expérimentales, répondant à différentes exigences émanant des utilisateurs, mais toutes sont compatibles avec les flots de conception industriels.

L'obtention d'un environnement complet permet au concepteur de spécifier la campagne d'injection, de contrôler l'exécution de la campagne par simulation ou émulation du circuit et d'obtenir une vue synthétique des résultats.

La génération d'un modèle détaillé de propagation d'erreurs est la première originalité de l'environnement proposé dans cette thèse, par comparaison avec les environnements précédents cités au chapitre 1. La deuxième originalité est l'utilisation de mutations optimisées pour l'instrumentation d'une description comportementale de circuit. Les mutants générés permettent

non seulement de réaliser des simulations au niveau RTL, d'où un gain notable de temps par rapport aux simulations niveau portes, mais ils prennent de plus en compte les contraintes spécifiques à l'émulation, mentionnées dans ce chapitre. Les chapitres 3 et 4 détaillent le processus de génération de ces mutants pour les différents modèles de fautes visés.

Chapitre 3.

Injection de fautes simples "SEU"

3.1. Limitation des approches précédentes de génération de mutants

La génération de mutants est étudiée depuis longtemps dans le domaine du test logiciel. Récemment, cette approche a été utilisée aussi dans le domaine du matériel afin de générer des séquences de tests fonctionnels pour des descriptions décrites en VHDL. Par exemple, dans [G.A. 96] et [G.A. 99], les "opérateurs mutés" incluent le remplacement d'opérateurs arithmétiques, de constantes et de variables, ou l'insertion d'opérateurs unaires. Des modifications similaires sont mentionnées dans [E.J. 94-b] pour réaliser des injections de fautes. Avec de telles modifications, le mutant peut en général être synthétisé si la description initiale est synthétisable. Cette approche peut donc être utilisée pour générer des mutants compatibles avec le processus de simulation ou d'émulation. Cependant, il est très difficile d'utiliser de tels mutants pour évaluer l'impact de fautes réelles pouvant survenir dans l'application car les "opérateurs mutés" ne sont pas directement liés à de telles fautes. L'analyse fonctionnelle de ce qui peut arriver suite à une mutation peut donc difficilement donner des informations utiles aux concepteurs.

Dans [A.F. 00] ou diverses autres publications similaires, l'injection d'erreurs dans les modèles VHDL est aussi considérée pour la génération de test fonctionnel. Ici, les modifications sont automatiquement introduites dans le code VHDL initial afin d'injecter des fautes de collage dans un ensemble d'emplacements donné. Les fautes sélectionnées sont activées pendant la simulation grâce à des entrées de contrôle primaires. Cela correspond à la génération de mutants pour un modèle de fautes simple, et pourrait souvent être remplacé par une simple insertion de saboteurs. Cependant cette approche est intermédiaire entre l'insertion classique de saboteurs (qui peuvent aussi être contrôlés par des entrées primaires) et la génération de mutants. L'injection de fautes autres que les collages n'est pas considérée.

Notre approche, appelée **génération contrôlée** de mutants est basée sur un procédé de génération différent. La génération des mutants (1) concerne des modèles de fautes précis associés aux fautes qui apparaissent dans l'application (et donc significatifs pour le concepteur) et (2) est optimisée pour la synthèse matérielle sur un émulateur (contraintes discutées dans la section 2.2.2). Dans la section suivante, nous proposons différents mutants générés pour la modélisation à haut niveau des effets de SEU sur les registres d'états.

3.2. Génération contrôlée de mutants pour l'injection de transitions erronées

Dans des descriptions niveau RTL, les registres d'état sont un cas particulier pour le processus d'injection. La conséquence d'une inversion de bit dans un tel registre peut être évaluée seulement si le codage des états est connu, c.-à-d. dans la plupart des cas après la synthèse du circuit et donc assez tard dans le processus de conception. Par contre, la modélisation de l'inversion de bit peut être utilisée sans aucun problème pour des variables ou des signaux qui correspondent après implémentation à des registres de données.

Dans la plupart des applications, les erreurs de données sont moins critiques que les erreurs de contrôle. Les données peuvent être protégées par des moyens de détection et correction tels que des codes ou des assertions dépendantes de l'application. De plus, l'impact sur l'application d'une erreur de donnée peut être très limité (ex. une valeur de pixel erronée dans une image). Dans le cas des erreurs de contrôle, le comportement du circuit peut a priori devenir imprévisible. Dans ce cas, l'analyse détaillée des conséquences est d'un intérêt majeur et une campagne d'injection de fautes dans des descriptions de haut niveau est une solution intéressante.

Il est alors nécessaire de modéliser à un niveau élevé l'effet d'un SEU qui survient dans un registre d'états. La solution proposée, présentée au chapitre 1, est d'injecter des transitions erronées asynchrones entre les états définis par leurs noms symboliques. Ce modèle de haut niveau a cependant deux inconvénients. D'abord, il n'est pas possible de savoir avant l'affectation des états (c.à.d. le codage du nom symbolique d'états en binaire) la correspondance entre les transitions erronées et les inversions de bits uniques. Le modèle proposé inclut donc des inversions de bit uniques et multiples, ce qui peut donner des résultats pessimistes si, dans le milieu opérationnel, il se produit uniquement des inversions de bit uniques. Par ailleurs, les transitions erronées sont en général injectées seulement entre les états existant dans la spécification du circuit (états valides). Les transitions vers des états invalides peuvent être prises en considération seulement si l'instruction "when others" a été spécifiée précisément pour prendre en compte les états invalides, mais ce n'est pas une pratique générale parce qu'elle limite les optimisations pendant la synthèse. Ce deuxième inconvénient signifie que l'effet d'un SEU conduisant à un état non spécifié ne peut pas être évalué avec précision dans la plupart des cas avant le codage final des états. Par exemple, dans le cas d'un codage compact, certains codes sont inutilisés si le nombre d'états n'est pas une puissance de 2. Une inversion de bit peut soit transformer un code légal en un autre code légal (saut erroné vers un autre état défini initialement dans la description du circuit) ou transformer un code légal en un autre code non utilisé. Ce dernier cas ne peut être analysé qu'une fois la description au niveau portes disponible. Par conséquent, il n'est pas inclus dans notre étude.

3.2.1. Différentes alternatives pour modifier une description VHDL

Différentes approches de génération de mutants pour l'injection de transitions erronées dans des machines à états (ci-après dénotées "FSM") ou des graphes de contrôle niveau RTL seront présentées et comparées dans cette section.

Le style de description le plus souvent utilisé (car efficace en synthèse) pour décrire une FSM inclut deux processus; le premier est séquentiel, sensible aux signaux d'horloge et d'initialisation et correspond à l'implémentation du registre d'états. Le deuxième est combinatoire et définit les fonctions logiques. Dans le processus combinatoire, la transition entre les états est souvent spécifiée dans une instruction "case" et la valeur des sorties est spécifiée soit sur les transitions soit sur les états. L'exemple de la Figure 3-1 représente un FSM à deux processus (séquentiel et combinatoire) avec un nombre d'états limité à trois.

```

entity FSM is
  port(
    -- inputs
    clk, reset : in std_logic ; -- clock and reset signals
    i1, i2 : in std_logic ;    -- primary inputs
    -- outputs
    o1, o2 : out std_logic); -- primary outputs
  end FSM;

architecture archi_FSM of FSM is
  type STATETYPE is (idlestate, init, doit) ;
  signal state, nstate: STATETYPE ;
begin
  seq: process(clk, reset)
  begin
    if reset='1' then state <= idlestate ;
    elsif (clk'event and clk='1') then
      state <= nstate ;
    end if ;
  end process ;

  comb: process (state, i1, i2)
  begin
    case state is
      when idlestate =>
        o1 <= '0' ;
        if (i1='0') then nstate <= idlestate ;
          o2 <= '0' ;
        else nstate <= init ;
          o2 <= '1' ;
        end if ;
    end case ;
  end process ;

  when init =>
    if (i2='1') then nstate <= idlestate ;
      o1 <= '0' ;
      o2 <= '0' ;
    elsif (i1='0') then nstate <= init ;
      o1 <= '0' ;
      o2 <= '1' ;
    else nstate <= doit ;
      o1 <= '1' ;
      o2 <= '0' ;
    end if ;
  when doit =>
    nstate <= idlestate ;
    o1 <= '1' ;
    o2 <= '1' ;
  end case ;
end process ;
end archi_FSM ;

```

Figure 3-1: Description VHDL d'un FSM simple

Quatre approches ont été initialement retenues pour modifier une telle description VHDL de sorte que des transitions erronées puissent être injectées pendant les expériences. Ces approches tiennent toutes compte des règles de synthèse liées aux outils Synopsys et Mentor (Leonardo). L'objectif est d'apporter le minimum de modifications possible dans la description initiale. La dernière contrainte consiste à définir des modifications systématiques afin qu'elles soient automatisables.

Pour ces approches, la description initiale est modifiée de façon à rendre possible l'injection de toutes les transitions erronées entre les états de la description initiale. On autorise l'injection par un signal d'entrée externe "en_inj" ajouté à l'entité ; il est alors possible de définir de manière *directe* la transition erronée à injecter, ou de la définir de manière *indirecte*. Dans le premier cas, l'état initial et final d'une transition ("I_state" et "F_state") correspondent à des signaux d'entrée insérés dans la définition de l'entité. Dans le deuxième cas, on indique l'état initial "I_state" et une valeur de déplacement "inj_N". L'état final de la transition "F_state" est déterminé comme étant l'état qui est à une distance "inj_N" de l'état "I_state" dans la liste des états énumérés spécifiée dans la description initiale. Ce type de définition de l'injection simplifie la commande externe lorsque toutes les transitions erronées doivent être injectées pendant une campagne. Dans les deux cas, il est possible soit de spécifier les états par des codes avec un nombre minimal de bits ou par les noms symboliques d'états. Si l'on choisi une spécification par codes, ces derniers seront assignés en fonction de l'ordre des états dans la définition du type énuméré ou suivant un codage introduit par l'utilisateur avant de lancer le processus de génération de mutants. En effet, dans certains cas il est possible d'utiliser le codage réel des états (s'il est disponible) et cela à des fins d'optimisation. Nous utiliserons les acronymes suivants afin d'identifier les quatre possibilités (alternatives) :

DFS ("Direct definition of Final State") : définition directe de l'état final.

IFS ("Indirect definition of Final State") : définition indirecte de l'état final.

SE ("State Encoding") : codage des états.

SN ("State symbolic Name") : noms symboliques des états.

3.2.1.1. DFS-SE

Toutes les versions "mutantes" permettent, par commande de signaux externes d'injection, de forcer les inversions souhaitées pour une expérience donnée. Dans le cas d'une combinaison DFS-SE, les entrées requises pour l'injection sont insérées à la fin de l'entité FSM comme décrit en Figure 3-2. La taille des vecteurs init_state_code et final_state_code est calculée en fonction du nombre total des états.

```

entity FSM is
  port
    (-- inputs
    clk, reset : in std_logic;    -- clock and reset signals
    i1, i2 : in std_logic;        -- primary inputs
    -- outputs
    o1, o2 : out std_logic;       -- primary outputs
    en_inj: in std_logic;         -- insertion
    init_state_code: in std_logic_vector(1 downto 0); -- insertion
    final_state_code: in std_logic_vector(1 downto 0) -- insertion
  );
end FSM;

```

Figure 3-2: Insertion des trois signaux d'entrées dans l'entité FSM définie en figure 3-1

Les codes spécifiés en signaux d'entrée avec un type classique (std_logic ou std_ulogic) doivent être convertis en noms symboliques. Ceci est réalisé par la fonction "std_to_state" représentée en Figure 3-3.

```

function std_to_state (signal S1: in std_logic_vector(1 downto 0)) return STATETYPE is
  begin
    case S1 is
      when "00" => return(idlestate);
      when "01" => return(init);
      when "10" => return(doit);
      when others => null;
    end case;
  end std_to_state;

```

Figure 3-3 : Fonction "std_to_state" générée pour l'exemple FSM

Au niveau de l'instruction case décrite dans la partie "architecture", chaque affectation de l'état suivant est remplacée par un appel de fonction "affect". Son rôle est d'injecter la transition erronée si le signal "en_inj" est mis à 1 et si l'état courant correspond à l'état initial de la transition erronée. La fonction "affect" appelle à son tour la fonction "std_to_state". À la place de cette fonction de conversion, il est possible d'utiliser un processus de conversion pour passer d'un signal d'entrée codé en binaire à un nom symbolique d'état. Plusieurs approches ont été comparées en utilisant un ensemble d'exemples et la meilleure solution en termes de surface et de chemin critique après synthèse est obtenue en général par la fonction "affect" avec un seul appel de la fonction "std_to_state" (comme illustré dans la Figure 3-4). La définition de cette fonction dépend du nombre d'états dans le FSM ou le graphe de contrôle à modifier. Donc la solution proposée consiste à remplacer, dans l'instruction case, toutes les affectations de type "nstate <= next_state_name" par "nstate <= affect(code_of_current_state, next_state_name, en_inj, init_state_code, final_state_code)", ce qui est assez simple à automatiser.

Remarque

Le paramètre effectif "code_of_current_state" de la fonction "affect" correspond au code de l'état qui est défini dans l'expression qui vient après le "when" du bloc "case".

```
function affect(state_code: std_logic_vector(1 downto 0);  
    nstate_name:    STATETYPE;  
    signal en_inj:    std_ulogic;  
    signal init_state_code: std_ulogic_vector(1 downto 0);  
    signal final_state_code: std_ulogic_vector(1 downto 0))  
    return STATETYPE is  
begin  
    if(en_inj='1' and init_state_code=state_code)then  
        return(std_to_state(final_state_code));  
    else  
        return(nstate_name);  
    end if;  
end affect;
```

Figure 3-4: Fonction "affect" générée pour l'exemple FSM

3.2.1.2. DFS-SN

La spécification de l'état initial et de l'état final d'une transition erronée en utilisant directement des noms symboliques a l'avantage d'éviter l'utilisation de la fonction de conversion et par conséquent d'obtenir un gain considérable en terme de logique combinatoire.

Cette fois, les entrées init_state_code et final_state_code sont déclarées de type STATETYPE comme illustré en Figure 3-5.

```
entity FSM is  
    port  
        ( -- inputs  
          clk, reset: in std_logic;  -- clock and reset signals  
          i1, i2: in std_logic;      -- primary inputs  
          -- outputs  
          o1, o2: out std_logic;    -- primary outputs  
          en_inj: in std_logic;    -- insertion  
          init_state_code: in STATETYPE;  -- insertion  
          final_state_code: in STATETYPE); -- insertion  
end FSM;
```

Figure 3-5: Insertion des trois signaux d'entrée dans l'entité FSM

Cependant, dans ce cas le type énuméré des états doit être défini dans un paquetage externe. Par conséquent, la définition de type est extraite de la description initiale et mise dans un nouveau paquetage. Par la suite une variable "state_tmp" est créée et toutes les affectations de "nstate" seront remplacées par des affectations de "state_tmp". L'injection est réalisée par une instruction "if-then-else" à la fin du processus ; cette instruction est illustrée en Figure 3-6.

```

if(en_inj='1' and init_state_code=state) then
    nstate<=final_state_code;
else
    nstate<=state_tmp;
end if;

```

Figure 3-6: Injection dans l'approche DFS-SN pour l'exemple FSM

Bien que la spécification de l'ensemble des signaux à injecter soit plus simple en utilisant des noms qui symbolisent les états, l'approche DFS-SN a quelques impacts sur le processus de synthèse. En effet, les deux entrées primaires du bloc (init_state_code et final_state_code) seront codées avec un nombre de bits correspondant au nombre de bits utilisé pour coder les états. Dans le cas d'un codage compact le résultat est similaire à celui obtenu en utilisant l'approche DFS-SE. Cependant, dans le cas d'un codage d'états de type 1 parmi N ("one-hot"), le nombre de bits peut être plus élevé. Ceci implique un sensible accroissement du nombre d'E/S dans le bloc, ce qui est un paramètre important quand on effectue une émulation matérielle.

3.2.1.3. IFS-SE et IFS-SN

Quand toutes les transitions erronées, ayant comme point de départ un état donné, doivent être explorées, les contrôles externes utilisés pour l'injection peuvent être simplifiés juste en définissant un déplacement par rapport à l'état courant. Ce déplacement définit l'état final désiré et il suffit juste de l'incrémenter pour injecter l'erreur suivante. Cependant, cette méthode implique plus de modifications dans le VHDL initial, à savoir l'utilisation de la fonction de conversion et le remplacement des affectations de "nstate" par des affectations de code binaire à une variable temporaire "state_tmp". Par ailleurs, l'utilisation de la valeur de déplacement implique l'implémentation de la fonction modulo. Cette fonction existe dans les paquetages standard mais étant donné qu'elle n'est définie que sur des entiers et que l'opérande gauche doit avoir une valeur égale à 2 puissance n, nous avons opté pour une description explicite illustrée dans la Figure 3-7. Ce bloc d'instructions est inséré à la fin du processus combinatoire. Ces modifications sont plus complexes que celles requises pour les approches DFS.

```

if(en_inj='1' and std_to_state(init_state_code)=state) then
    tmp_add := std_logic_vector(unsigned(inj_N)+unsigned(state_tmp));
    if (tmp_add > "10") then
        tmp_add := std_logic_vector(unsigned(tmp_add)+"01");
    end if;
else
    tmp_add:=state_tmp;
end if;
    nstate<=std_to_state(tmp_add);

```

La différence entre le nombre maximum d'états codés sur 2 bits et le nombre réel des états (4-3=1).

Figure 3-7: Utilisation explicite du modulo pour le mode IFS-SE

Le bloc d'instructions présenté en Figure 3-7 concerne plus précisément le mode IFS-SE. L'opérande "01" de la 2ème affectation de "tmp_add" est utilisée afin d'éviter un état non défini dans la description initiale.

Les modifications nécessaires au mode IFS-SN ne sont pas très différentes de celles requises pour l'approche IFS-SE. En effet, en IFS-SN nous devons définir le type énuméré dans un paquetage externe, changer le type dans la déclaration du signal "init_state" et remplacer le deuxième appel de la fonction "std_to_state" par un bloc de conversion de type "case". Ceci est illustré dans la Figure 3-8.

```

if (en_inj='1' and init_state=state) then
    tmp_add := std_logic_vector(unsigned(in_N)+unsigned(state_tmp));
    if (tmp_add > "10") then
        tmp_add := std_logic_vector(unsigned(tmp_add)+"01");
    end if;
else
    tmp_add := state_tmp;
end if;
case tmp_add(1 downto 0) is
    when "00" => nstate<=idlestate;
    when "01" => nstate<=init;
    when "10" => nstate<=doit;
    when others => null;
end case;

```

Figure 3-8: Utilisation explicite du modulo pour le mode IFS-SN

L'avantage de ces deux approches (définition indirecte de la transition erronée) par rapport à l'approche DFS se situe au niveau du calcul de l'état final. En effet, avec le mode IFS, nous sommes sûr de tomber sur un état final erroné (du moment que "in_N" est supérieur à 0) alors qu'une définition directe de la transition erronée peut parfois être similaire à une transition attendue. Cependant, l'utilisation des opérations d'addition dans les approches IFS implique une augmentation de la logique combinatoire par rapport aux approches DFS. Ceci n'a pas réellement de conséquences pour des campagnes à base de simulation, mais peut avoir des conséquences sur l'émulation en termes de surface et de fréquence maximale de fonctionnement.

3.2.2. Généralisation de l'étude sur des registres d'état de type entier

Le codage des états d'un FSM peut être défini préalablement par le concepteur. Ce dernier a alors la possibilité de déclarer ces états soit de type entier soit de type bit_vecteur. Dans le cas d'une définition de type entier des états, l'approche de génération des mutants étudiée précédemment est facilement généralisable. Pour les approches de génération de mutants DFS-SN et IFS-SN, la spécification directe de l'état initial et de l'état final d'une transition erronée implique une insertion de signaux d'entrées de type entier, ce qui est toléré par l'outil de synthèse. Par contre, la génération des mutants en suivant les modes DFS-SE et IFS-SE consiste

a adapter la fonction de conversion vue précédemment pour passer cette fois-ci d'un signal "bit_vecteur" vers un signal "entier" (Figure 3-9). Elle doit aussi prendre en compte le codage défini dans la description initiale.

```
-- Définition des états dans la description initiale
constant idlestare : integer := 2;
constant init      : integer := 1;
constant doit      : integer := 0;

-- Définition de la fonction de conversion (presque similaire à la fonction std_to_state)

function std_to_state (signal S1:in std_logic_vector(1 downto 0) )
    return INTEGER is
begin
    case S1 is
        when "10" => return(idlestare);
        when "01" => return(init);
        when "00" => return(doit);
        when others => null;
    end case;
end std_to_state;
```

Figure 3-9: Fonction de conversion "std_to_state"

3.2.3. Evaluation qualitative des surcoûts de mutation sur des exemples de circuit

Tous les modes de mutation présentés ci-avant sont pris en compte par l'outil développé et ont été utilisés sur une partie du jeu de test ITC99, dont certains exemples comportent des registres d'état définis par des entiers. Le Tableau 3-1 résume les résultats obtenus [R.L. 00]. Il montre l'ordre de grandeur des surcoûts obtenus après synthèse par l'outil Leonardo de Mentor Graphics (en utilisant la bibliothèque de cellules standards "cub" d'AMS). Ceci permet d'évaluer les surcoûts induits par le processus de mutation par rapport aux trois paramètres principaux pertinents dans le cas d'une émulation (complexité matérielle, fréquence d'horloge maximale et nombre d'entrées/sorties), en fonction des deux procédures principales de codage des états ("one-hot" : une bascule active par état, et "compact" ou "binaire" : un nombre minimum de bascules implémentées pour le registre d'états). N'importe quelle autre technologie peut évidemment être choisie pendant la synthèse des mutants.

Le nombre d'E/S dépend fortement de l'exemple, et donc seules des marques qualitatives sont données (de "+" qui veut dire bon ou surcoût faible jusqu'à "--" très mauvais ou surcoût élevé). Ces résultats montrent clairement l'impact du style de modification VHDL sur les caractéristiques d'un circuit muté et chargé sur émulateur. La définition du processus de mutation est donc décisive quant à l'efficacité des campagnes d'injection de fautes basées sur l'émulation. Les différents modes de mutation permettant la réalisation des mêmes injections, il sera dans la plupart des cas préférable d'utiliser une approche de type DFS-SN avec codage compact des états.

Tableau 3-1: Résumé des résultats obtenus à partir des benchmarks ITC99 pour la génération de mutants (surcoût en complexité, diminution de la fréquence d'horloge et augmentation du nombre d'E/S par rapport à la description initiale) [R.L. 00]

Codage des états	1-parmi-N ("One-hot")			Compact (binaire)		
Approches	Complexité	Fréquence	E/S	Complexité	Fréquence	E/S
DFS-SE	10%-150%	(0%)-(−60%)	+	2%-200%	(0%)-(−10%)	+
DFS-SN	10%-20%	(0%)-(−50%)	--	2%-10%	(0%)-(−10%)	+
IFS-SE	15%-100%	(−10%)-(−60%)	+	5%-50%	(−5%)-(−40%)	+
IFS-SN	15%-80%	(−5%)-(−50%)	-	15%-50%	(−5%)-(−40%)	+

3.2.4. Utilisation d'informations partielles de synthèse

Certaines données issues de la synthèse peuvent être prises en considération à une étape intermédiaire du flot de conception, avant que la "netlist" optimisée au niveau portes ne soit produite. En particulier, le codage des états peut être disponible bien avant la génération de la "netlist" finale. Des codes d'état réels peuvent alors être utilisés pour prendre en compte uniquement les transitions erronées correspondant à des inversions de bit uniques ; ceci conduit à l'amélioration de l'analyse des modes de défaillance pour ce modèle. Il est à préciser que les résultats des expériences d'injection de fautes exécutées avant que le codage des états ne soit disponible peuvent être réutilisés. Il suffit d'écarter, pendant l'analyse (c'est-à-dire la génération du modèle comportemental), les données qui ne sont plus significatives. Par conséquent, le modèle épuré sera obtenu très rapidement. En outre, il n'est pas obligatoire de générer une version de mutant qui est limitée au codage des états effectif. Dans la pratique, un mutant généré pour injecter toutes les transitions erronées possibles peut être utilisé pour n'importe quel type de codage d'état. Dans le cas où le codage est défini assez tôt, la campagne d'injection peut être écourtée en injectant seulement les transitions erronées qui sont significatives.

Une fois que le codage des états est disponible ou lorsque des injections doivent être réalisées dans des registres de données, des inversions de bits uniques peuvent aussi être réalisées avec un autre type de mutants. Ceci exige une décomposition de la valeur du registre en bits commandés indépendamment pendant les expériences d'injection. Une telle mutation a été également automatisée et permet au concepteur de faire des expériences en tenant compte de l'effet des transitions erronées vers des états invalides (c'est-à-dire des états qui correspondent à des codes d'état inutilisés). Cela n'est pas possible avec l'injection des transitions erronées dans des descriptions de haut niveau. Les différentes modifications VHDL nécessaires pour effectuer l'injection sont précisées dans la section suivante.

3.3. Génération de mutants pour les inversions de bits

Dans le cas des inversions de bits uniques, utilisées pour modéliser les SEUs à un niveau d'abstraction inférieur à celui des transitions erronées (vu précédemment), les différentes modifications et insertions apportées à la description initiale sont illustrées en gras en Figure 3-10.

Un bloc d'instruction "case" inséré dans la partie séquentielle nous permet d'inverser un des points mémoires du registre "state". Le bit à inverser est sélectionné par un signal d'entrée externe "Sel" ajouté à l'entité. Le même type de modification peut être utilisé pour effectuer des inversions de bits dans des registres de données. Nous montrons dans la Figure 3-10 uniquement la partie affectation car le reste des modifications qui concerne le signal d'horloge sera discuté dans la section suivante.

Les résultats de génération du modèle comportemental obtenus avec les deux types de mutants seront comparés dans le chapitre 5.

```

signal state: std_logic_vector(2 downto 0) ;
...
if(En_inj='0') then
    state <= nstate ;
else case Sel is
    when "01" => state(0) <= not state(0) ;
    when "10" => state(1) <= not state(1) ;
    when "11" => state(2) <= not state(2) ;
    when others => null;
    end case;
end if;

```

Figure 3-10: Application du modèle des inversions de bits uniques sur le registre "state".

- **Extension aux éléments mémoires de type entier**

Le modèle de fautes "inversion de bits" peut être appliqué uniquement sur des éléments mémoires de type binaire. Les cibles d'injection qui sont déclarées initialement de type non binaire (de type entier par exemple) doivent être converties pour être prises en compte. Par conséquent, nous devons effectuer une conversion explicite en utilisant la fonction suivante : **CONV_type_vers_lequel_on_veut_convertir**(élément_à_convertir).

Un exemple de conversion de type est illustré par la Figure 3-11. Le signal "Elemty" de type entier est choisi comme cible. Pour pouvoir effectuer des inversions de bits, nous devons insérer dans la description initiale un signal intermédiaire binaire "Elemt_biny". La taille de ce dernier est calculée en fonction de la borne supérieure "n". Une fois l'inversion effectuée, la valeur finale obtenue après injection et contenue dans le vecteur "Elemt_bin_INJy" est reconvertie en entier.

```

signal Elemty: INTEGER range 0 to n ; -- Cible d'injection de type entier.

signal Elemt_biny:STD_LOGIC_VECTOR(Ly downto 0) ; -- Insertion d'un signal intermédiaire
-- de type binaire (avec Ly= $\lceil \log_2(n) \rceil$ ).

...
Elemt_biny <= CONV_STD_LOGIC_VECTOR(Elemty, Ly) ; -- Conversion de type.

... -- Application des processus d'injection sur des éléments de type binaire.

Elemty <= CONV_INTEGER(Elemt_bin_INJy) ; -- Récupérer le résultat d'injection
-- et le convertir en entier.

```

Figure 3-11: Exemple de conversion de type.

3.4. Contrôle externe des instants d'injection

Nous allons présenter dans cette section la modélisation, au niveau RTL, de l'aspect asynchrone relatif au phénomène SEU dont l'instant d'apparition est indépendant de l'horloge. Il s'agit d'étendre les modifications vues précédemment afin de permettre au concepteur de réaliser des injections qui se rapprochent de la réalité opérationnelle du circuit cible.

La solution proposée consiste à ajouter un processus, illustré par la Figure 3-14, afin d'obtenir un contrôle externe du registre d'états par le signal "Asyn_inj". Les signaux de contrôle "En_inj" et "Asyn_inj" ajoutés comme entrées primaires permettent de réaliser des injections asynchrones, représentant l'effet des SEU. L'implémentation du processus correspond à une porte XOR (Figure 3-14). Le signal "Asyn_inj" (combiné avec le signal d'horloge nominal "Clk") constitue l'entrée primaire qui nous permet de contrôler l'horloge "Clk". Le nouveau signal d'horloge "Clk_reg" est généré en sortie.

Le protocole peut également être illustré par les chronogrammes des Figures 3-12 et 3-13. Il nécessite deux signaux de contrôle "Asyn_inj" et "En_inj". Le signal "En_inj" est utilisé afin de sélectionner les entrées de certains éléments mémoires du circuit, ciblés par l'injection, de façon à modifier le traitement par rapport à celui réalisé en mode de fonctionnement normal. Cela dépend non seulement du signal "En_inj" mais aussi de l'adresse de la cible.

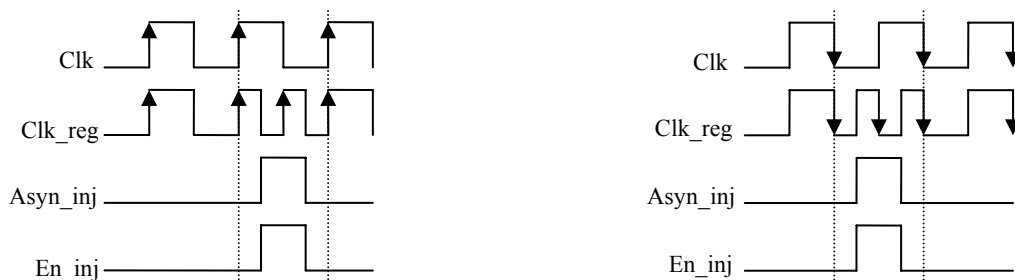


Figure 3-12: Protocole d'injection (injection active sur front montant ou descendant)

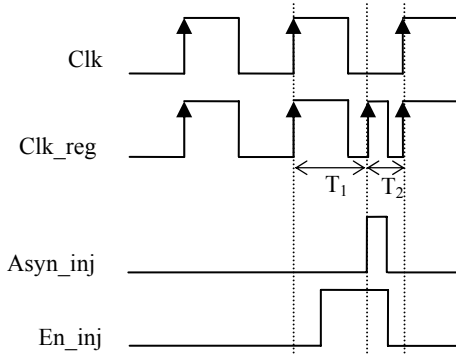


Figure 3-13: Injection au milieu de la 2^{ème} moitié de la période

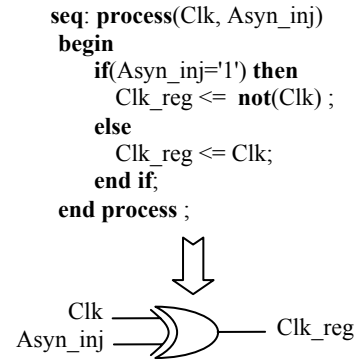


Figure 3-14: Génération du signal Clk_reg

La porte XOR de la Figure 3-14 assure correctement la fonctionnalité recherchée. Cela consiste à rendre possible la génération d'un nouveau front d'horloge à n'importe quel instant situé entre les deux fronts d'horloge d'un cycle donné. En effet, le passage du signal "Asyn_inj" de 0 à 1 implique l'inversion de la valeur du signal "Clk_reg", ce qui correspond, dans la Figure 3-13, à l'instant d'injection. Au contraire, dans les chronogrammes de la Figure 3-12, l'injection se produit au milieu de la période. Cependant, pour le bon déroulement d'une injection asynchrone, il est important de respecter les deux contraintes suivantes :

- Les deux signaux de contrôle "Asyn_inj" et "En_inj" sont remis à 0 avant le front d'horloge qui suit l'injection.
- Le signal "En_inj" est actif (au niveau haut) uniquement au front d'horloge relatif à l'injection.

Dans le cas d'une campagne d'injection basée sur la simulation post-synthèse avec délais ou l'émulation, les délais T_1 et T_2 (illustrés dans la Figure 3-13) doivent durer suffisamment longtemps pour que les signaux à l'entrée des registres puissent se stabiliser avant l'impulsion d'injection et aussi avant la montée suivante du front d'horloge. Par conséquent, la fréquence de l'horloge de la simulation ou du prototype doit être choisie de telle sorte que le délai T_2 soit supérieur au chemin critique. Par ailleurs, si le concepteur souhaite réduire au minimum le temps nécessaire pour exécuter les expériences d'injection alors la fréquence d'horloge maximum sera obtenue en effectuant des injections de fautes au milieu de la période et dans ce cas, la fréquence d'horloge de la simulation ou du prototype peut être égale à (fréquence nominale)/2.

Afin de rendre asynchrone le contrôle de l'injection décrite dans l'exemple de la section précédente (Figure 3-10), il suffit de remplacer le signal d'horloge nominal par le nouveau signal d'horloge "Clk_reg" (Figure 3-15).

```

signal state: std_logic_vector(2 downto 0) ;

seq: process(clk_reg, Reset)
begin
  if Reset='1' then state <= waiting ;
  elsif (clk_reg'event and clk_reg='1') then
    if (En_inj='0') then
      state <= nstate ;
    else case Sel is
      when "01" => state(0) <= not state(0) ;
      when "10" => state(1) <= not state(1) ;
      when "11" => state(2) <= not state(2) ;
      when others => null;
    end case;
  end if;
end if;
end process ;

```

Figure 3-15: La partie séquentielle d'un FSM muté (injection asynchrone pour le modèle des inversions de bits uniques).

Les signaux "En_inj" et "Asyn_inj" nous permettent d'obtenir un contrôle externe de l'instant d'injection. Pendant une expérience de simulation ou d'émulation, le programme de test, qui se charge d'appliquer les entrées fonctionnelles au circuit, va aussi agir sur les signaux de contrôle externes afin d'exécuter l'injection ; ceci nous offre la possibilité d'effectuer plusieurs injections pendant une seule expérience (multiplicité temporelle supérieur à 1). Par ailleurs, nous discuterons l'aspect multiplicité spatiale dans le chapitre 4.

3.5. Extension aux mémoires RAM et bancs de registres

Une RAM est fonctionnellement équivalente à un groupement de "n" registres de taille "m". Chaque registre possède "m" bascules avec une entrée de validation pour chacune d'elle. Lorsqu'il y a écriture ou lecture d'un mot, c'est l'ensemble des bascules du registre qui sont activées. En VHDL RTL, une RAM ou un banc de registres est généralement défini par un tableau de vecteurs mémorisés. La Figure 3-16 représente un exemple VHDL d'une mémoire de 4 mots de deux bits chacun.

Les modifications VHDL apportées à la description initiale concernent particulièrement le processus séquentiel modélisant l'écriture de données dans le banc de registres cible (Figure 3-17). Nous définissons la cible d'injection par le biais des signaux de sélection "Sel_word" et "Sel_bit" insérés dans la définition de l'entité. Le mot cible est déterminé par le sélecteur "Sel_word" et le numéro du bit cible est défini par l'entrée de sélection "Sel_bit". Des instructions sont insérées dans la partie conditionnelle qui exprime le front d'horloge. L'activation des signaux "En_inj" et "En_asyn" permet d'exécuter l'instruction d'affectation de la valeur obtenue après inversion de bit. La variable "word_inj", insérée dans la description

initiale, sera utilisée comme variable intermédiaire afin de permettre l'inversion de bit. Ceci s'effectue après exécution de l'instruction "mem(Sel_word)<= word_inj".

```

entity RamMemory is
    port (data : in std_logic_vector(1 downto 0);
          address : in std_logic_vector(1 downto 0);
          we, clk : in std_logic;
          q : out std_logic_vector(1 downto 0)
          );
    end RamMemory;

architecture example of RamMemory is
    type MemType is array(3 downto 0) of
        std_logic_vector (1 downto 0);
    signal mem : MemType;
begin

    process (clk, we, address)
    begin
        if (clk='1' and clk'event)then
            if we='1' then
                mem(conv_integer(address)) <= data;
            end if;
        end if;
    end process;

    process (clk, address)
    begin
        if (clk='1' and clk'event) then
            q <= mem(conv_integer(address));
        end if;
    end process;
end example;
    
```

Figure 3-16: Description d'une RAM qui regroupe 4 mots de 2 bits chacun

```

entity RamMemory is
    port (data : in std_logic_vector(1 downto 0);
          address : in std_logic_vector(1 downto 0);
          we, clk : in std_logic;
          q : out std_logic_vector(1 downto 0);
          En_inj : in std_logic;
          En_asyn : in std_logic;
          -- Selection du mot cible
          Sel_word : in integer range 0 to 3;
          -- Numero du bit a inverser
          Sel_bit : in integer range 0 to 1);
    end RamMemory;

architecture example of RamMemory is
    type MemType is array(3 downto 0) of
        std_logic_vector (1 downto 0);
    signal mem : MemType;
    signal Clk_reg : std_logic; -- memory clock
begin
    process (clk_reg, we, address, Sel_word, Sel_bit)
    variable word_inj : std_logic_vector(1 downto 0);
    begin
        if (clk_reg='1' and clk_reg'event)then
            if en_inj='1' then
                word_inj := mem(Sel_word);
                word_inj(Sel_bit) := not word_inj(Sel_bit);
                mem(Sel_word) <= word_inj;
            elsif we='1' then
                mem(conv_integer(address)) <= data;
            end if;
        end if;
    end process;

    process (clk_reg, address)
    begin
        if (clk_reg='1' and clk_reg'event) then
            q <= mem(conv_integer(address));
        end if;
    end process;
    ----- insertion de process -----
    process (En_asyn, Clk)
    begin
        if (En_asyn='1') then
            Clk_reg <= not(Clk);
        else Clk_reg <= Clk;
        end if;
    end process;
end example;
    
```

Figure 3-17: Modification de la RAM pour des inversions de bit uniques

3.6. Modifications structurelles induites par les mutants

Nous avons élaboré jusqu'ici des méthodes de modification de descriptions VHDL qui permettent, d'une part, de réaliser des injections simples en désignant une seule cible parmi tous les éléments mémoire, et d'autre part, de choisir parmi les différents modèles de fautes celui qui convient à cette cible (en fonction du type d'élément ciblé : registre, registre d'états, RAM ... et du niveau d'abstraction considéré).

Nous allons étudier dans cette partie l'impact de l'implémentation des modèles de fautes "inversion de bits" et "transition erronée" au niveau architectural. Il s'agira par la suite (chapitre 4) d'élaborer une méthode de modification de circuit globale qui permet de réaliser des injections dans plusieurs éléments cibles (hétérogènes).

3.6.1. Transitions erronées

Cette partie concerne uniquement des registres de contrôle avec des états symboliques (le codage des états n'est pas encore connu). Nous commençons par décrire le cas le plus simple : injection dans un seul registre d'états.

La Figure 3-18 donne le schéma d'un FSM avant modification. "R_state" représente le registre d'états, tandis que les signaux CS et NS correspondent respectivement à l'état courant et à l'état suivant. L'architecture d'un FSM inclut deux parties. La première est séquentielle, sensible aux signaux d'horloge et d'initialisation et correspond à l'implémentation du registre d'états. La seconde est combinatoire ; elle est utilisée pour déterminer les transitions entre les états et la valeur des sorties (spécifiées soit sur les transitions soit sur les états).

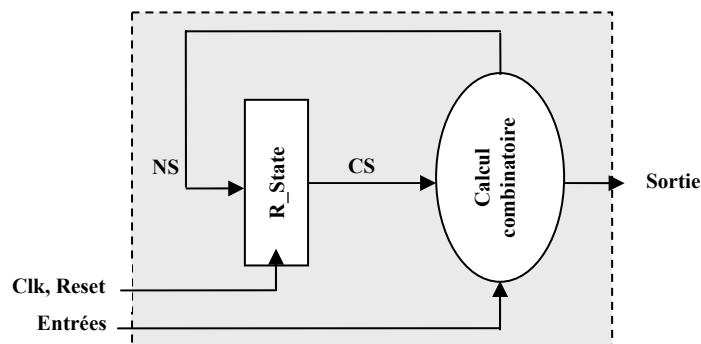


Figure 3-18: Machine d'états avant modification

Pour pouvoir injecter des transitions erronées dans le registre d'états "R_state", un multiplexeur est ajouté avant l'entrée du registre "R_state" (Figure 3-19). Il est contrôlé par le signal d'entrée externe "En_inj" et permet lorsque ce dernier est à "1" d'introduire la valeur de "Inj_S" dans le registre. Sinon c'est la valeur de l'état suivant "NS" qui est assignée à l'entrée du registre d'états (fonctionnement en mode nominal).

Comme nous l'avons déjà mentionné, dans le cas d'un modèle de fautes de type transition erronée, les états sont définis comme noms symboliques groupés en type énuméré. Ainsi, pour déterminer la transition erronée à injecter "Inj_S", il faut ajouter au circuit une fonction de conversion. En effet, pour les approches DFS-SE et IFS-SE, les valeurs assignées au nouveau vecteur d'entrée "Sel" sont de type binaire et doivent être converties en noms symboliques (selon l'ordre des états dans la définition du type énuméré).

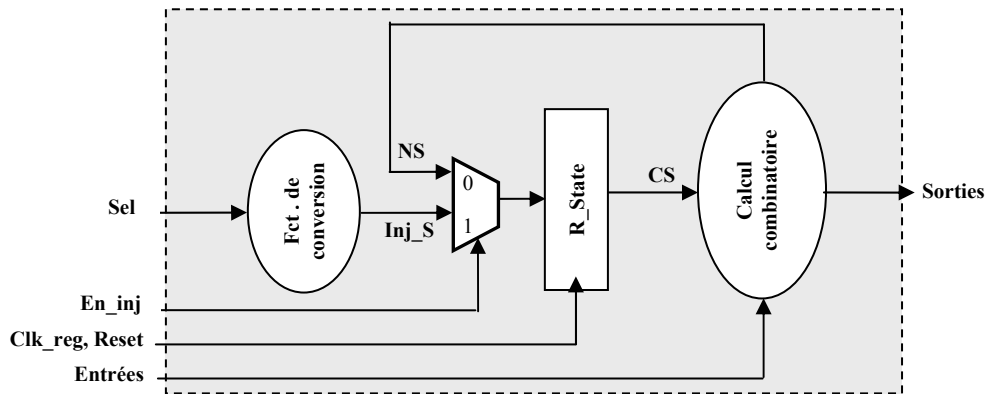


Figure 3-19: Machine d'états après modification

3.6.2. Inversion de bits pour les registres de donnée ou de contrôle

Les registres sont composés d'éléments mémoires sensibles à un front (bascule D, bascule JK ...). La synthèse logique d'un circuit qui implémente le modèle de fautes "inversion de bits" pour corrompre un registre de donnée ou un registre de contrôle (après codage des états) inclut l'insertion de multiplexeurs à l'entrée du registre cible "Ri" (Figure 3-20). Le multiplexeur est contrôlé par le nouveau signal d'entrée "En_inj" et permet lorsque ce dernier est à "1" d'introduire dans le registre la donnée "In_inj_i" obtenue après inversion. Sinon c'est la donnée "In_i", issue d'un fonctionnement nominal (logique combinatoire "LC_i"), qui est reliée à l'entrée du registre cible. Le circuit instrumenté inclut également des blocs combinatoires "LC_i bit flip" insérés à la sortie des registres. Ils sont composés généralement d'inverseurs et de multiplexeurs. Ces blocs permettent, en fonction de l'adresse indiquée pour le vecteur d'entrée "Sel", d'inverser le contenu de la bascule cible.

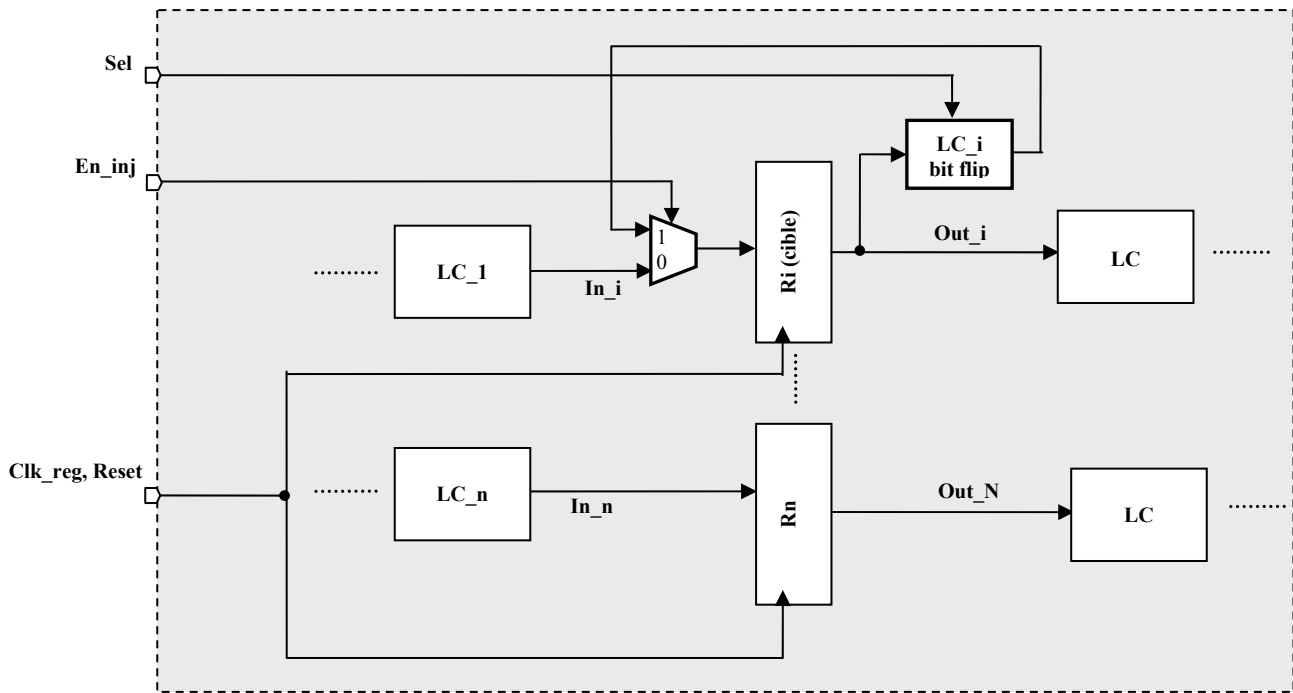


Figure 3-20: Inversion de bits dans un registre cible "Ri"

3.6.3. RAM ou banc de registres

Dans la section 3.5, nous avons appliqué le modèle des inversions de bits uniques sur un exemple VHDL correspondant au comportement d'une mémoire décrite par un tableau de vecteurs mémorisés de 4x2. La synthèse au niveau RTL de l'exemple illustré par la Figure 3-17 a été réalisée par l'outil de synthèse Leonardo. Elle génère un schéma RTL dont l'interprétation fait ressortir deux fonctions logiques illustrées par la Figure 3-21. L'ensemble de la logique combinatoire additionnelle est confiné dans des blocs spécifiés en pointillés.

La première fonction est la **"Fonction d'adressage instrumentée"**. Le rôle de cette fonction consiste à activer la ligne du banc de registres adressée, selon la valeur du signal **"En_inj"**, par l'une des deux entrées d'adressage. Dans le cas d'un fonctionnement nominal, la valeur d'adressage sera assignée par l'entrée **"Address"**. Le signal **"we"** détermine le mode d'accès (lecture ou écriture). Dans le cas d'un fonctionnement en mode d'injection, l'accès au banc de registres est exclusivement réservé en mode écriture et la valeur d'adressage sera assignée par l'entrée **"Sel_word"**. Dans cette fonction logique, les éléments insérés dans le circuit initial correspondent à un décodeur 1/n et un ensemble de "n" multiplexeurs dont chacun est connecté aux entrées de validation des "m" bascules (banc de "n" registres de "m" bits).

La deuxième fonction appelée **"Fonction d'inversion"** est dédiée à l'inversion du bit de donnée contenu dans l'un des registres et ciblé par les sélecteurs **"Sel_word"** et **"Sel_bit"**. La ligne cible est déterminée par le sélecteur **"Sel_word"** qui contrôle deux multiplexeurs connectés au bus de donnée en sortie du banc de registres. Par ailleurs, le numéro de bit cible est défini par

l'entrée de sélection "Sel_bit". Ainsi, nous obtenons à l'entrée du banc de registres des bits de donnée dont un seul est inversé.

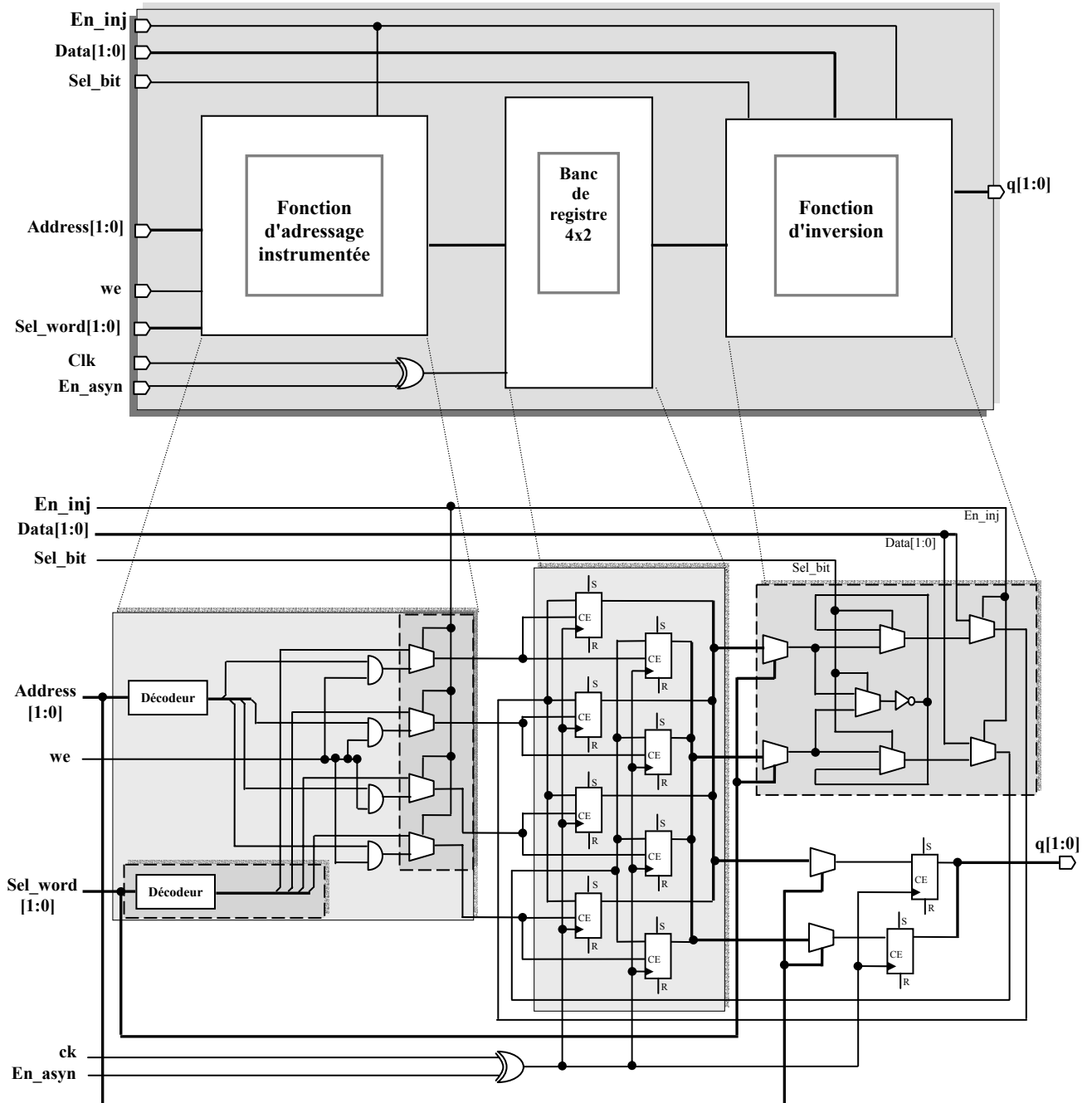


Figure 3-21: Représentation structurelle du surcoût induit par l'instrumentation du banc de registres

Nous présentons dans le Tableau 3-2 l'impact des modifications sur des bancs de registres de tailles différentes (n/m : n registres de m bits). Le tableau montre des surcoûts obtenus après synthèse par l'outil Leonardo de Mentor Graphics (en utilisant la bibliothèque de cellules standards "cub" d'AMS). Le coût de la modification est évalué en terme de nombre d'E/S ajouté, nombre de cellules obtenues après synthèse et chemin critique dans la netlist générée.

Les résultats montrent que le surcoût en termes de surface et de vitesse tend à décroître lorsque la taille du banc de registres augmente.

Tableau 3-2: Surtout en complexité, fréquence d'horloge et augmentation du nombre d'E/S par rapport à la description initiale d'un banc de registres

Taille du banc de registres	E/S	Surface	# instances	Chemin critique
4/2	+5	+42,9%	+59,1%	+107%
8/4	+7	+42,2%	+77,5%	+175,5%
16/8	+9	+30,9%	+59%	+72,3%
256/16	+14	+23,7%	+46,8%	+87,9%

3.6.4. Le cas des registres non ciblés par l'injection

Pendant le cycle d'injection, à l'exception du registre cible, tous les registres du circuit doivent être maintenus en mode de mémorisation. Lors de la synthèse logique du modèle muté, une entrée de validation (appelée "ENA" et qui est active à 1) sera ajoutée au niveau de tous les registres qui ne sont pas ciblés par l'injection (Figure 3-22); ces derniers sont désactivés lorsque la valeur du signal "En_inj" est égale à 1 (mode injection). La Figure 3-23 présente le cas d'un registre qui comportait une entrée de validation avant l'instrumentation du circuit cible.

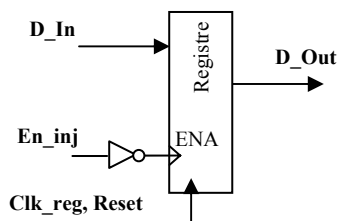


Figure 3-22: Ajout d'une entrée pour assurer la mémorisation

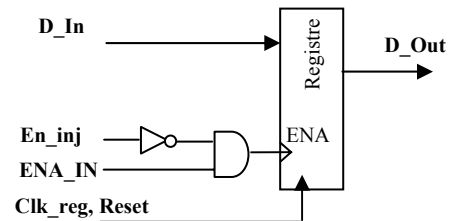


Figure 3-23: Combinaison entre le signal d'injection et l'entrée de validation

Au niveau RTL, cela correspond à retranscrire les différentes instructions d'affectation non concernées par l'injection dans la partie conditionnelle où le signal "En_inj" est inactif. La Figure 3-24-(b) illustre les différentes modifications apportées à la partie séquentielle d'un FSM lorsque seul le registre "state" est choisi comme cible d'injection ; les autres registres "R_i" seront inclus dans la première partie conditionnelle contrôlée par le signal "En_inj".

<pre> signal state, nstate: std_logic_vector(2 downto 0) ; seq: process(clk, Reset) begin if Reset='1' then state <= waiting ; elsif (clk'event and clk='1') then state <= nstate ; R_i <= ...; ...; end if ; end process ; </pre> (a)	<pre> signal state, nstate: std_logic_vector(2 downto 0) ; seq: process(clk_reg, Reset) begin if Reset='1' then state <= waiting ; elsif (clk_reg'event and clk_reg='1') then if (En_inj='0') then state <= nstate ; R_i <= ...; ...; else case Sel is when "01" => state(0) <= not state(0) ; when "10" => state(1) <= not state(1) ; when "11" => state(2) <= not state(2) ; when others => null; end case; end if ; end if ; end process ; </pre> (b)
---	--

Figure 3-24: Retranscription des instructions d'affectation des registres non ciblés par l'injection

Nous représentons dans la Figure 3-25 la synthèse logique de l'exemple muté décrit en partie dans la Figure 3-24-(b).

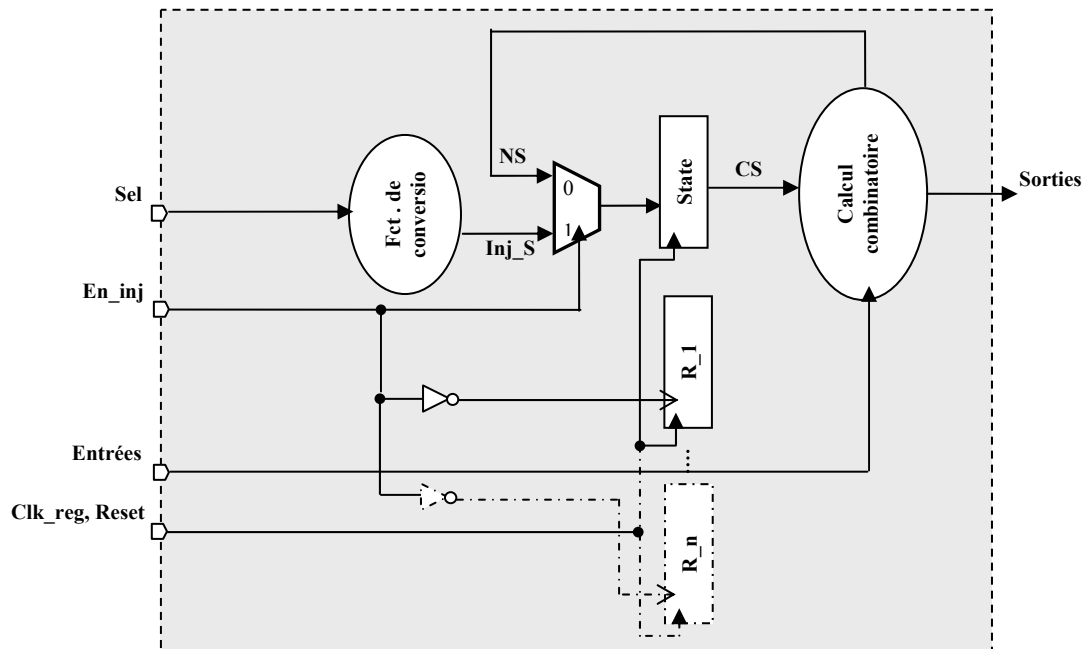


Figure 3-25: Insertion des entrées de validation pour les registres non ciblés par l'injection

3.7. Conclusion

Nous avons présenté dans ce chapitre les différents mutants générés permettant d'effectuer des injections de fautes simples dans les points mémoire du circuit, tout en étant synthétisables de façon efficace. Ils permettent de plus de prendre en compte plusieurs niveaux de modélisation des fautes. Nous avons également démontré la faisabilité de l'automatisation de la génération contrôlée de mutants VHDL pour l'injection des transitions erronées. Les résultats ont prouvé que le type d'instrumentation a un impact important sur une campagne d'injection basée sur l'émulation, car les différentes modifications peuvent mener à des caractéristiques de prototypage du circuit très différentes.

Jusqu'ici, les mutants générés prenaient en compte le modèle des inversions de bit uniques et le modèle des transitions erronées afin de représenter à différents niveaux d'abstraction le phénomène SEU. Nous avons présenté dans la section 1.2.1.6 l'intérêt d'étendre les modèles de fautes développés jusque là au modèle MBF pour répondre aux besoins d'analyse dans certains contextes. Dans le chapitre suivant, nous allons spécifier des mutants plus complexes, permettant d'injecter des fautes avec une multiplicité spatiale définie par l'utilisateur.

Chapitre 4.

Injection de fautes multiples "MBF"

La plupart des expériences d'injection de fautes rapportées dans la littérature utilisent le modèle des inversions de bits uniques. Cependant, ce modèle ne peut pas être directement utilisé dans certains cas. En effet, l'évolution des technologies rend ce modèle de moins en moins représentatif. Les SEUs représentent un type de fautes transitoires parmi d'autres fautes qui sont susceptibles de se produire dans un circuit. Les types de fautes qui sont à l'origine du modèle MBF et qui justifient l'extension du modèle SEU sont présentés dans la section 1.2.1.

Il devient donc nécessaire de se préoccuper de la conséquence d'erreurs transitoires sur des bits multiples. Il faut noter qu'une telle multiplicité était seulement indirectement prise en compte, dans un cas particulier, par le modèle des "transitions erronées", décrit précédemment.

Nous discutons dans ce chapitre l'extension du modèle "inversions de bits uniques" au modèle "inversions de bits multiples". Ensuite, nous présenterons une nouvelle approche d'instrumentation d'une description VHDL au niveau RTL ; elle permettra à un concepteur d'injecter aussi bien des inversions de bits uniques/multiples que des transitions erronées. Cette génération hétérogène de mutant peut être, en outre, configurée en fonction de la généralité exigée du modèle de fautes en termes de localisation des fautes multiples.

4.1. Génération hétérogène des mutants

Dans la plupart des descriptions de haut niveau certains registres sont directement assignés à des valeurs binaires (par exemple les registres de données) et d'autres sont assignés à des valeurs symboliques. Ces valeurs ne seront assignées à des valeurs binaires que lors de la synthèse. C'est la première raison qui justifie l'intérêt d'une modélisation hétérogène, c'est-à-dire d'un processus de mutation assurant la compatibilité entre l'injection de plusieurs types de fautes. Ainsi, il est possible d'injecter des transitions erronées là où c'est nécessaire et des inversions de bits uniques ou avec une multiplicité donnée dans les autres éléments de mémorisation du circuit.

L'autre raison qui justifie la modélisation hétérogène est le besoin croissant de considérer un plus grand nombre d'origines de fautes. Par exemple, comme il a été détaillé dans la section 1.2.1, la propagation d'un SET dans la logique peut avoir comme conséquence la modification simultanée de plusieurs bits. La solution d'étendre le modèle de fautes habituel pour réaliser des

injections de MBFs dans une description niveau RTL permet d'évaluer très tôt l'impact de telles configurations d'erreur sur l'application. Evidemment, cela implique un temps supplémentaire pour exécuter les expériences d'injection de fautes ; des compromis sont donc à définir entre le temps d'expérimentation et la précision de l'analyse. Il est également possible de tenir compte d'informations de plus bas niveau si celles-ci sont disponibles. Dans le cas de la réutilisation d'un bloc, la description niveau portes de ce bloc peut être disponible ainsi que le résultat d'une analyse par un outil semblable à celui décrit dans [D.A. 02] ; il est alors possible de limiter les injections aux configurations d'erreur observées sur les sorties du bloc. Si quelques options de synthèse spécifiques seront utilisées plus tard dans le flot de conception (par exemple en limitant la multiplicité d'erreurs de sortie comme dans [N.T. 97]), il est également possible d'en tenir compte pendant la spécification de la campagne d'injection de fautes.

Dans le cas des MBFs, un autre choix est à faire, lié au type de phénomène que l'on souhaite modéliser. La modélisation la plus générale doit supposer que les bits erronés qui sont multiples peuvent être répartis partout dans le circuit (là où il y a des cellules de mémorisation). Par exemple, ceci peut arriver lorsqu'un flux très intense de particules frappe le circuit. Quand on regarde d'autres sources de fautes (par exemple les problèmes d'intégrité du signal), des erreurs simultanées de bit peuvent également se produire dans plusieurs registres qui n'ont aucune relation fonctionnelle particulière. Par contre, dans le cas de la propagation d'un SET, les bits erronés peuvent apparaître uniquement dans un ensemble limité de registres, connectés sur les sorties du bloc combinatoire. Et quand on considère un rayon laser muni d'une grande focalisation, les erreurs peuvent se limiter à un seul registre, particulièrement si quelques précautions sont prises pendant le placement et le routage du circuit. Nous illustrons dans la Figure 4-1 trois attaques par laser sur une zone de registres placée et routée selon deux options différentes. Dans la configuration (a), deux registres sont touchés avec une multiplicité par élément égale à 2. Dans la configuration (b), quatre registres sont touchés avec une multiplicité par élément égale à 1. Tandis que dans la configuration (c) où le diamètre de focalisation est plus grand, quatre registres sont touchés avec une multiplicité maximale égale à 2.

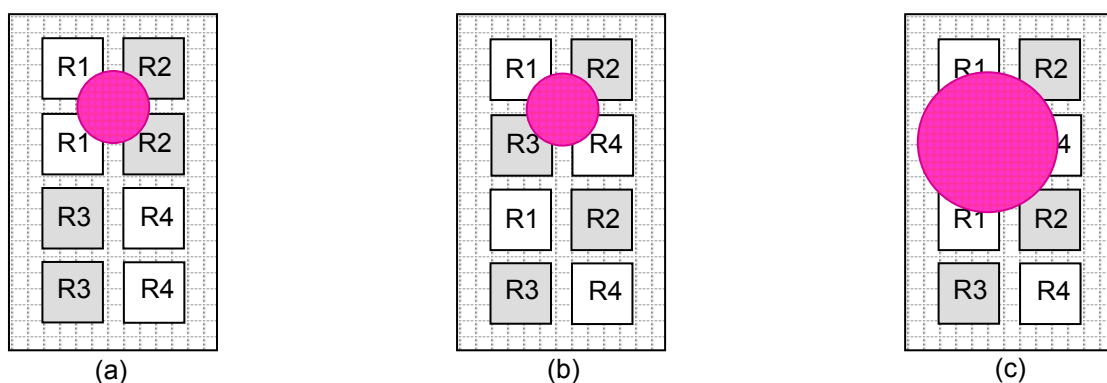


Figure 4-1: Exemples d'attaques par laser selon (a) un placement/routage 1 et une focalisation 1 (b) un placement/routage 2 et une focalisation 1 (c) le placement/routage 2 et une focalisation 2.

Le flot d'analyse global illustré par la Figure 2-2 (section 2.2.1) peut tout à fait être employé pour étudier la robustesse d'un circuit face à de telles attaques par faute. Il suffit en effet de définir les événements redoutés comme étant ceux permettant à un pirate d'obtenir des informations significatives. Il faut cependant s'assurer que les types de fautes injectés sont réellement représentatifs des phénomènes induits par une attaque.

4.2. Les différentes alternatives pour modifier une description VHDL

Pour qu'un mutant soit le plus général possible, il devrait permettre à un concepteur d'injecter n'importe quel nombre d'inversions de bits (inférieur à une multiplicité donnée) dans n'importe quel ensemble d'éléments de mémorisation d'un circuit donné. Pour atteindre cet objectif, nous proposons une solution de mutation automatisable. Cependant, cette approche qui se veut générale implique un inconvénient en terme de complexité et par conséquent en terme de coût d'implémentation (particulièrement si un prototypage matériel est utilisé comme support pour la campagne d'injection de fautes). Selon certaines hypothèses, une modification moins générale, limitant ainsi les injections multiples à des sous-ensembles d'éléments de mémorisation, peut être suffisante. Cette solution sera également présentée, limitant les configurations possibles d'erreur, mais réduisant la complexité des modifications VHDL.

4.2.1. Architecture globale

Il est donc nécessaire de pouvoir générer des mutants plus complexes, permettant d'injecter des fautes avec une multiplicité spatiale définie par l'utilisateur. Par ailleurs, la prise en compte du cas particulier des transitions erronées doit pouvoir être transparente pour l'utilisateur. Il s'agit donc de générer un mutant pour la description globale du circuit, avec des modèles de fautes pouvant être différents d'un bloc à l'autre dans la hiérarchie. La génération de tels mutants hétérogènes a été automatisée, selon le principe résumé en Figure 4-2.

Pour une expérience d'injection donnée, le bloc ciblé est sélectionné dans le cas où les fautes multiples sont confinées dans un sous-bloc. Dans le cas le plus général, le décodeur serait supprimé au prix d'un plus grand nombre d'entrées externes ajoutées. Les indices des bits à inverser sont définis sur des signaux externes (avec une multiplicité égale ou inférieure à la multiplicité maximum définie pour l'ensemble de la campagne d'injections). Les éléments mémoire définis comme cible d'injection peuvent être regroupés dans des registres "virtuels" au niveau des blocs du circuit. Cette approche, présentée en détail dans la section 4.2.3, permet de cibler simultanément plusieurs registres appartenant à un bloc donné. Si le bloc sélectionné contient une machine à états spécifiée avec un type énuméré, les informations nécessaires pour injecter une transition erronée sont transmises par les mêmes ports externes et stockés également dans le registre virtuel. Dans la section qui va suivre, l'approche d'injection se limite à un seul registre. L'extension à un sous bloc sera présentée en section 4.2.3.

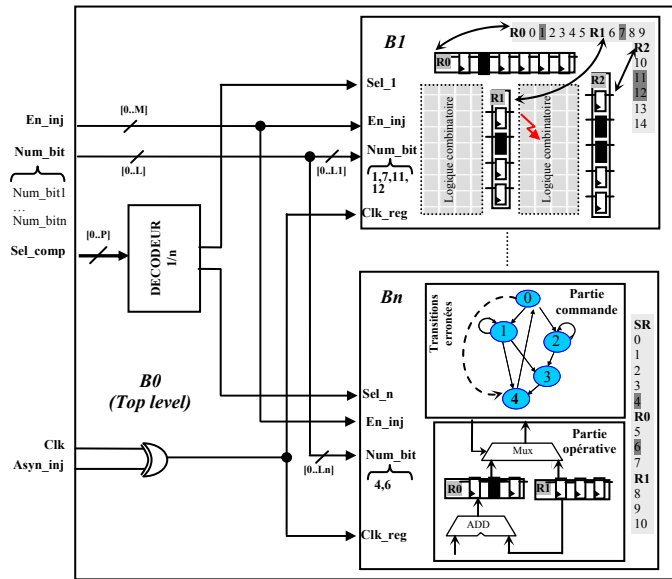


Figure 4-2: Architecture globale d'un mutant pour modèle de fautes hétérogène

4.2.2. Injections multiples limitées à un seul registre

Nous allons présenter dans cette section les différents mutants VHDL générés pour l'injection des MBFs. Nous allons nous limiter dans un premier temps à un seul registre cible.

Les entrées de contrôle requises pour l'injection sont insérées au niveau du bloc cible comme illustré dans la Figure 4-3. Nous représentons également dans la figure les différentes valeurs associées aux entrées de contrôle. Le signal "Sel_reg" permet de désigner le registre sur lequel l'inversion multiple sera effectuée. La taille de ce signal dépend du nombre total des registres concernés par l'injection.

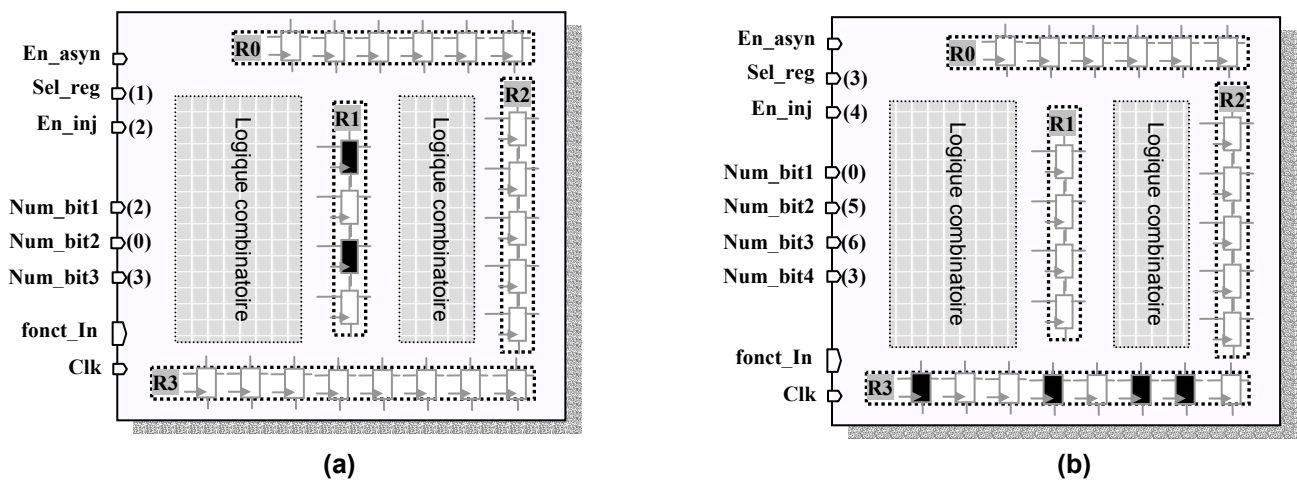


Figure 4-3 : Injection localisée de MBF dans les registres (a) R1 et (b) R3, respectivement de multiplicité deux et quatre

Les entrées : "Num_bit1", "Num_bit2", ... "Num_bitn", appelées sélectionneurs de bit, correspondent aux numéros de bit utilisés pour désigner les bits du registre à inverser. Elles sont déclarées de type entier sur un intervalle borné (dans le but d'optimiser la taille des entrées car par défaut, la taille d'un signal non borné est implémenté sur 32 bits par les différents outils de synthèse). Par ailleurs, la taille des sélectionneurs de bit est définie en fonction de la taille du plus grand registre cible, de telle sorte à pouvoir agir sur tous ses bits.

L'entrée de contrôle "En_inj" permet de définir la multiplicité (le nombre de bits à inverser simultanément). En effet, elle permet de déterminer l'ensemble des sélectionneurs de bit ("Num_bitn") qui seront pris en compte lors de l'injection multiple. Par exemple, lorsque la valeur du signal "En_inj" est égale à L alors les endroits cibles du registre (désigné par "Sel_reg") seront définis par l'ensemble des entrées : "Num_bit1", "Num_bit2", ... "Num_bitL". Seuls les numéros de bits appartenant à cet ensemble seront effectivement inversés (erreur de multiplicité L).

➤ En VHDL

Nous allons décrire dans la suite les différentes étapes pour obtenir une version VHDL mutée. La Figure 4-4 illustre l'insertion des différentes entrées requises pour l'injection dans la partie entité.

```
entity my_entity is
  port ( Clk , reset : std_logic;
        ...
        -- Ports d'entrée/sortie nécessaires pour l'injection
        En_inj : in std_logic_vector(i downto 0);      -- Sélection du nombre de bits à inverser
        En_asyn : in std_logic;                        -- Contrôle asynchrone de l'injection
        Sel_reg : in std_logic_vector(t downto 0);    -- Sélection du registre cible
        Num_bit1 : in integer range 0 to m-1;         -- Numéro du 1er bit à inverser
        Num_bit2 : in integer range 0 to m-1;         -- Numéro du 2ème bit à inverser
        ...
        Num_bitn : in integer range 0 to m-1          -- Numéro du nème bit à inverser
  );
end my_entity;
```

i : fonction de la borne supérieure de la représentation binaire du nombre de bits à inverser
t : fonction de la borne supérieure de la représentation binaire du registre choisi pour l'injection.
m : fonction de la taille du plus grand registre

Figure 4-4: Insertion des signaux de contrôle dans l'entité.

• Modification des processus séquentiels

Les modifications illustrées dans la Figure 4-5 seront effectuées au niveau de chaque processus séquentiel modélisant un ou plusieurs registres cibles définis par le concepteur. Les modifications concernent plus particulièrement les instructions d'affectation relatives aux registres cibles. Ces derniers seront en fonctionnement nominal (sans injection de fautes) lorsque tous les bits de l'entrée primaire "En_inj" sont à zéro. Dans le cas contraire

(fonctionnement en mode d'injection), la valeur contenue dans l'entrée de sélection "Sel_reg" déterminera l'instruction d'affectation " $R_i \leq R_i_Inj$ " qui sera effectivement exécutée ; elle peut éventuellement cibler l'instruction d'affectation " $Reg_state \leq Reg_state_inj$ " dans le cas d'injection de transitions erronées dans une machine d'états. La nouvelle valeur obtenue après injection (" R_i_Inj " ou " Reg_state_inj ") est transmise à l'élément mémoire associé (" R_i " ou " Reg_state "). Ces instructions sont insérées dans la partie conditionnelle qui exprime le front d'horloge.

```

process(Clk_reg, Reset, En_inj)
begin
  if Reset = '1' then
    ... -- Valeurs d'initialisation
  elsif (Clk_reg'event and Clk_reg = '1') then
    if En_inj = "bits à zéro" then
      ... -- mise à jour nominale des différents registres
    elsif (Sel_reg = "sélecteur du registre R1") then
      R1 <= R1_inj;
    elsif (Sel_reg = "sélecteur du registre R2") then
      R2 <= R2_inj;
      ...
    elsif (Sel_reg = "sélecteur du registre Rn") then
      Rn <= Rn_inj;
    elsif (Sel_reg = "sélecteur du registre d'état") then
      Reg_state <= Reg_state_inj;
    end if;
  end if;
end process ;

```

Figure 4-5: Modification du processus séquentiel.

- **Insertion du processus combinatoire pour l'inversion multiple**

Le processus qui se charge d'inverser les contenus des bascules cibles (décrit dans la Figure 4-6) est inséré à la fin de la description initiale. Avant de réaliser l'inversion, il convient de récupérer les valeurs des registres à injecter. Ensuite, les inversions de bit sont spécifiées par les instructions d'affectation de type " $R_i_inj(num_bit_i) \leq \text{not } R_i(num_bit_i)$ ". L'indice " num_bit_i " correspond à une des entrées primaires " $num_bit_1 \dots num_bit_n$ ". Ainsi, il nous permet d'obtenir un contrôle externe sur les points mémoires à inverser. Cette expression n'est autorisée que si l'indice est déclaré de type entier. Les éléments intermédiaires " R_i_inj " associés à chaque registre cible sont déclarés de façon globale aux différents processus qui composent la description. Les instructions d'affectation relatives à ces éléments intermédiaires sont répétées pour chaque sélecteur " num_bit_i ". L'exécution de chaque ensemble sera contrôlée par des conditions qui utilisent l'entrée de contrôle " En_inj ".

L'instruction "case" insérée à la suite de l'instruction d'affectation " $Rn_inj(num_bit1) \leq \text{not } Rn(num_bit1)$ " permet d'injecter des transitions erronées. En effet, dans le cas où le registre cible (choisi par le signal "Sel_reg") correspond à un registre de contrôle, l'état final erroné sera directement injecté par le biais du signal " num_bit1 ".

```

process (En_inj, R1, R2, ..., Rn)
begin
  R1_inj <= R1;
  R2_inj <= R2;
  ...
  Rn_inj <= Rn;

  if En_inj="1 bit à injecter" or En_inj="2 bits à injecter" or ... En_inj="n bits à injecter" then
    R1_inj(num_bit1) <= not R1(num_bit1);
    R2_inj(num_bit1) <= not R2(num_bit1);
    ...
    Rn_inj(num_bit1) <= not Rn(num_bit1);
    case num_bit1 is
      when 0 => Reg_state_inj <= nom_symb_0;
      when 1 => Reg_state_inj <= nom_symb_1;
      when 2 => Reg_state_inj <= nom_symb_2;
      ...
      when n => Reg_state_inj <= nom_symb_n;
      when others => null;
    end case;
  end if;

  if En_inj="2 bits à injecter" or ... En_inj="n bits à injecter" then
    R1_inj(num_bit2) <= not R1(num_bit2);
    R2_inj(num_bit2) <= not R2(num_bit2);
    ...
    Rn_inj(num_bit2) <= not Rn(num_bit2);
  end if;

  ...
  if En_inj="n bits à injecter" then
    R1_inj(num_bitn) <= not R1(num_bitn);
    R2_inj(num_bitn) <= not R2(num_bitn);
    ...
    Rn_inj(num_bitn) <= not Rn(num_bitn);
  end if;
end process;

```

Figure 4-6: Insertion du processus combinatoire qui réalise l'inversion multiple.

4.2.3. Injections multiples sans limitation spatiale à l'intérieur d'un sous-bloc

La méthode de génération de mutants vue précédemment permet d'effectuer des inversions multiples localisées sur un seul registre. Nous proposons, dans ce qui va suivre, d'étendre l'approche afin d'offrir aux concepteurs la possibilité de réaliser des injections multiples sur plusieurs cibles à la fois. Les bits concernés par les inversions simultanées peuvent se répartir librement au niveau des éléments spécifiés dans la liste des cibles d'injection. Ceci implique la génération d'un processus VHDL spécifique, regroupant dans un registre virtuel les différents éléments de mémorisation cibles.

Avec cette approche, l'exemple de la Figure 4-7 montre qu'il est possible de viser deux types d'applications. En effet, les bits erronés dans la Figure 4-7(a) touchent seulement deux registres et leur apparition se concentre dans une zone limitée ; ceci peut modéliser une attaque par fautes lors d'un tir laser. Par contre dans la Figure 4-7(b), les bits erronés sont répandus

aléatoirement et surviennent dans trois registres ; ceci peut correspondre à la mémorisation de plusieurs bits erronés due à la propagation d'un SET dans la logique combinatoire.

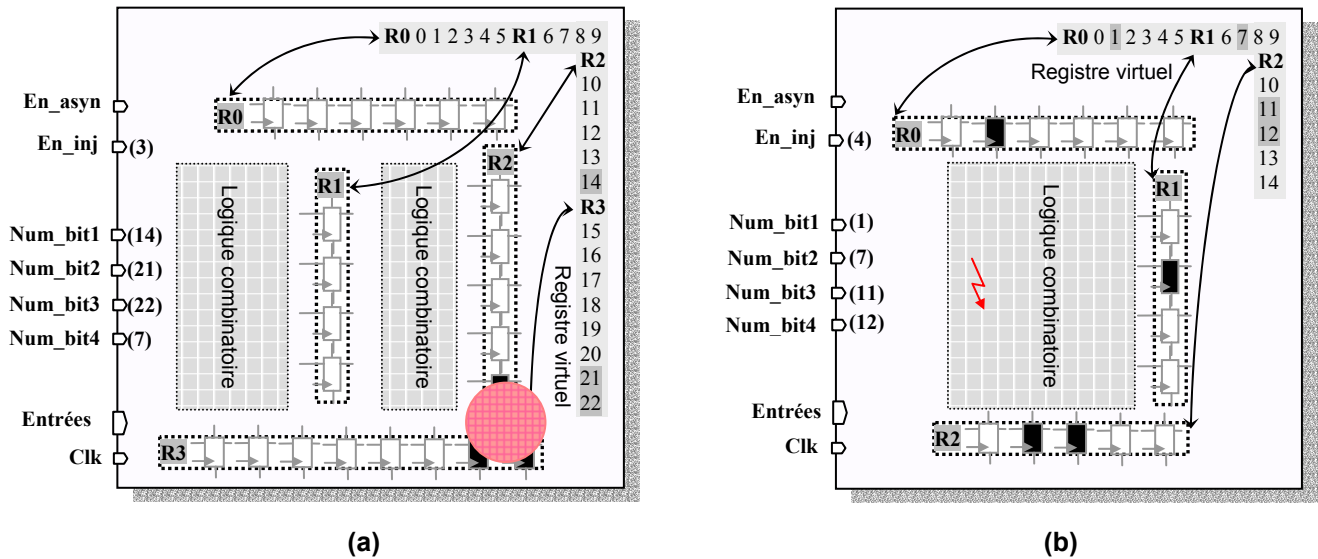


Figure 4-7 : Injection généralisée de MBF par le biais du registre virtuel

➤ En VHDL

Nous allons décrire dans la suite les différentes étapes pour obtenir une version VHDL mutée. La Figure 4-8 illustre l'insertion des différentes entrées requises pour l'injection dans la partie entité. Les endroits de l'injection multiple sont spécifiés par les entrées externes suivantes : "num_bit1", "num_bit2", ... "num_bitn". Elles permettent de designer 1) le numéro de bit à inverser dans le cas d'un modèle de fautes inversion de bits et/ou 2) l'état final d'une transition "F_state" lorsqu'il s'agit d'injecter des transitions erronées (dans un registre d'états).

```

entity My_entity is
  port
    ( -- inputs/outputs
      clk, reset : in std_logic ; -- Signaux d'horloge et d'initialisation
      ... --control inputs insertion
      En_inj : in std_logic_vector(i downto 0); -- Sélection du nombre de bits à inverser
      En_asyn : in std_logic; -- Contrôle asynchrone de l'injection
      num_bit1 : in integer range a to b; -- Numéro du 1er bit à inverser
      num_bit2 : in integer range a to b; -- Numéro du 2ème bit à inverser
      ....
      num_bitn : in integer range a to b -- Numéro du nème bit à inverser
    );
end My_entity;

```

Figure 4-8: Insertion des signaux de contrôle dans l'entité.

• Modification des processus séquentiels

De même que l'approche précédente, les modifications touchent les processus séquentiels modélisant un ou plusieurs registres cibles (Figure 4-9). Les instructions d'affectation des

nouvelles valeurs obtenues après injection sont insérées dans la partie conditionnelle qui exprime le front d'horloge. Cependant, lors de l'activation de l'injection ("En_inj" différent de zéro), il n'y a pas de contrôle sur les registres. Toutes les instructions d'affectation " $Elem_t_i \leq Elem_t_i_inj$ " sont exécutées.

```
seq: process( Clk_reg, Reset, En_inj)
begin
  if Reset='1' then
    Elem_t_0 <= ... ;    -- Affectation d'initialisation
    ...
    Elem_t_i <= ... ;
  elsif (Clk_reg'event and Clk_reg='1') then
    if En_inj = "tous les bits à 0" then
      Elem_t_0 <= ... ;    -- Mise à jour des registres
      ...                  -- Fonctionnement nominal
      Elem_t_i <= ... ;
    else
      Elem_t_0 <= Elem_t_0_inj; -- Fonctionnement en
      ...                      -- mode injection
      Elem_t_i <= Elem_t_i_inj;
      Reg_state <= Reg_state_inj;
    end if;
  end if;
end process ;
```

Figure 4-9: Modification du processus séquentiel.

```
seq: process( Clk_reg, Reset, En_inj)
begin
  if Reset='1' then
    Elem_t_0 <= ... ;    -- Affectation d'initialisation
    ...
    Elem_t_i <= ... ;
  elsif (Clk_reg'event and Clk_reg='1') then
    if (Sel_x='1') then
      if En_inj = "tous les bits à 0" then
        Elem_t_0 <= ... ;    -- Mise à jour des registres
        ...                  -- Fonctionnement nominal
        Elem_t_i <= ... ;
      else
        Elem_t_0 <= Elem_t_0_inj; -- Fonctionnement en
        ...                      -- mode injection
        Elem_t_i <= Elem_t_i_inj;
        Reg_state <= Reg_state_inj;
      end if;
    end if;
  end if;
end process ;
```

Figure 4-10: Modification du processus séquentiel dans le composant d'une description hiérarchique.

Une des limites de cette approche est liée à l'injection spatiale dans le cas d'une description hiérarchique. En effet, à l'instant d'injection un seul bloc du circuit est concerné par l'injection multiple. Pour étendre cette technique d'injection et celle qui précède aux descriptions hiérarchiques, les registres des blocs qui ne sont pas ciblés par l'injection passent en mode de mémorisation grâce au signal de sélection "**Sel_x**" (Figure 4-10). Lorsque la valeur de ce signal passe à 0, le contenu des éléments de mémorisation est conservé. Plus de détails sont donnés dans la section 4.3.

- **Insertion du processus d'inversion multiple commun à tous les éléments mémoires**

Le nouveau processus, décrit dans la Figure 4-11, sera inséré à la fin de la description initiale. Ce processus se charge d'inverser les contenus des bascules cibles. Ces dernières choisies sans limitation spatiale sont référencées par les entrées "num_bit1...num_bitn" ajoutées comme entrées primaires.

La variable "**All_reg_tmp**" correspond à un registre virtuel. Elle est utilisée pour regrouper, à l'aide de l'opérateur de concaténation binaire "&", le contenu initial de tous les registres cibles " $Elem_t_i$ ". L'inversion est réalisée par des instructions d'affectation de type " $All_reg_tmp(Num_bitn) := not All_reg_tmp(Num_bitn)$ ". La valeur du signal de contrôle "**En_inj**" permet de déterminer le nombre d'inversions simultanées de bits à effectuer.

L'utilisation de la variable "All_reg_tmp" permet de simplifier la modélisation de l'injection. En effet, la complexité du processus, et en particulier le nombre d'instructions "if", dépend du nombre d'inversions maximal (défini par le signal "En_inj") et non pas du nombre total des registres cibles. Une partie du registre virtuel inclut l'information relative au cas particulier des transitions erronées.

Les affectations définies à la suite de l'instruction "case" permettent de récupérer les valeurs associées aux vecteurs "Elemti". Ces valeurs doivent passer, en premier lieu, par des vecteurs intermédiaires "Elemti_Inj". Les instructions d'affectation "Elemti <= Elemti_Inj", insérées au niveau des processus séquentiels (Figure 4-10), réalisent la transmission explicite des valeurs des éléments "Elemti_Inj" aux éléments "Elemti" associés. Le processus de la Figure 4-11 génère une structure logique combinatoire dépourvue d'éléments mémorisants (bascules ou verrous).

```
process (En_inj, num_bit1, ..., num_bitn, Elemt0, ..., Elemti)
variable All_reg_tmp : std_logic_vector(1 downto 0);
begin
    All_reg_tmp := Elemti ... & Elemt1 & Elemt0;
    if En_inj = "1 bit à injecter" or En_inj = "2 bits à injecter" or ... En_inj = "n bits à injecter" then
        All_reg_tmp(num_bit1) := not All_reg_tmp(num_bit1);
    end if;
    if En_inj = "2 bits à injecter" or ... En_inj = "n bits à injecter" then
        All_reg_tmp(num_bit2) := not All_reg_tmp(num_bit2);
    end if;
    ...
    if En_inj = "n bits à injecter" then
        All_reg_tmp(num_bitn) := not All_reg_tmp(num_bitn);
    end if;

    Elemti_inj <= All_reg_tmp(1 downto 1-li);
    ...
    Elemt1_inj <= All_reg_tmp(n downto n-l1);
    Elemt0_inj <= All_reg_tmp(n-l1-1 downto 0);
end process;
```

Figure 4-11: Insertion du processus qui réalise l'inversion multiple.

- **Extension de l'approche aux transitions erronées**

Nous proposons d'étendre l'approche vue précédemment afin d'injecter dans le même circuit des "inversions de bits multiples" et des transitions erronées.

Nous gardons les mêmes entrées de contrôle "en_seu", "en_inj" et de sélection "num_bit1...num_bitn". Les sélecteurs de bit contribuent non seulement à injecter des "inversions de bits multiples" mais aussi des transitions erronées. Par rapport au processus de la Figure 4-11, les nouvelles modifications sont spécifiées en gras dans le processus de la Figure 4-12.

Les quelques adaptations apportées à la stratégie d'injection précédente consistent à :

- 1) Insérer une suite de "L" zéros dans la chaîne d'éléments utilisée pour construire le signal "All_reg_tmp". Cette suite de zéros est relative au registre d'états "Reg_state" (et la taille "L" correspond au nombre des états du FSM). Ce dernier prend comme valeur des noms symboliques d'états "nom_symb_0, ..., nom_symb_L". Dans le cas d'une injection de type transition erronée, l'état final est obtenu en appliquant le numéro correspondant sur l'une des entrées "num_bit_i". La correspondance entre les numéros et les noms symboliques d'états "nom_symb_0, ..., nom_symb_L" est obtenu par rapport à la position des zéros dans le registre virtuel "All_reg_tmp" et l'ordre des états dans la définition du type énuméré.
- 2) Ajouter à la fin du processus une instruction "case" qui sera contrôlée par le signal "All_reg_tmp(x downto x-L)". Cet intervalle correspond à la suite de 0 réservée au registre d'états. Son contenu qui est éventuellement ciblé par l'injection doit être converti en nom symbolique d'états. Ceci est réalisé par l'instruction "case".

```

process (En_inj, num_bit1, ..., num_bitn, Elemt0, ..., Elemti)
  variable All_reg_tmp : std_logic_vector(N downto 0);
  begin
    All_reg_tmp := Elemti ... & "tous les bits à 0" & ... & Elemt1 & Elemt0;

    if En_inj = "1 bit à injecter" or En_inj = "2 bits à injecter" or ... En_inj = "n bits à injecter" then
      All_reg_tmp(num_bit1) := not All_reg_tmp(num_bit1);
    end if;
    if En_inj = "2 bits à injecter" or ... En_inj = "n bits à injecter" then
      All_reg_tmp(num_bit2) := not All_reg_tmp(num_bit2);
    end if;
    ...
    if En_inj = "n bits à injecter" then
      All_reg_tmp(num_bitn) := not All_reg_tmp(num_bitn);
    end if;

    Elemti_inj <= All_reg_tmp(N downto N-li);
    ...
    Elemt1_inj <= All_reg_tmp(n downto n-l1);
    Elemt0_inj <= All_reg_tmp(n-l1-1 downto 0);

    case All_reg_tmp(x downto x-L) is
      when "00..001" => Reg_state_inj <= nom_symb_0;
      when "00..010" => Reg_state_inj <= nom_symb_1;
      when "00..100" => Reg_state_inj <= nom_symb_2;
      ...
      when "100..00" => Reg_state_inj <= nom_symb_n;
      when others => Reg_state_inj <= Reg_state;
    end case;
  end process;
end process;

```

Figure 4-12: Insertion du processus qui réalise l'inversion multiple.

4.2.4. Modifications structurelles induites par les mutants

Dans le cas d'une mutation de type MBF, les modifications VHDL apportées à la description initiale sont synthétisables. Nous étudions dans cette section les conséquences de ces modifications après avoir effectué une synthèse logique. Pour faciliter l'identification des éléments logiques additionnels, nous appliquons une instrumentation pour des inversions multiples sans limitation spatiale sur un exemple VHDL simple ; il décrit un échantillonnage par le biais des entrées fonctionnelles "DataIn1" et "DataIn2" (Figure 4-13).

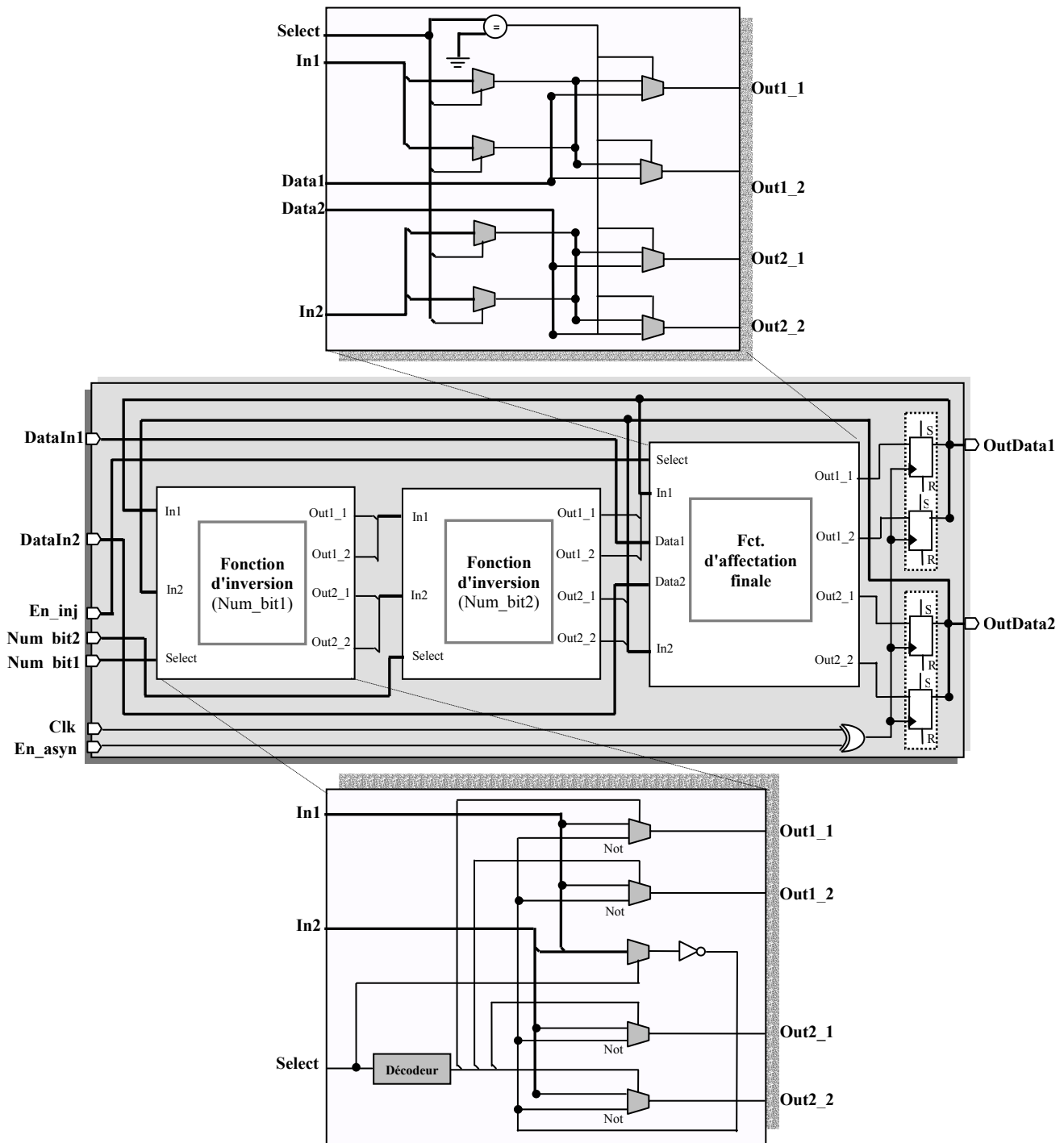


Figure 4-13: Représentation structurelle des modifications pour des injections de type MBF

Les données sont mémorisées dans deux registres de deux bits chacun. La synthèse au niveau RTL de cette description mutée, réalisée par l'outil de synthèse "Leonardo", génère un schéma RTL dont l'interprétation fait ressortir les fonctions logiques illustrées par la Figure 4-13. Deux fonctions logiques sont identifiées :

- La première fonction est redondante : "Fonction d'inversion (Num_bit1)" et "Fonction d'inversion (Num_bit2)". Le rôle de cette fonction consiste à inverser un seul des bits constituant les entrées In1 et In2. Le numéro de bit cible est défini par l'entrée "select". Le bloc "**Fonction d'inversion (Num_bit1)**" se charge d'inverser, selon la valeur de l'entrée primaire "Num_bit1", un des bits de donnée "DataOut1" ou "DataOut2" constituant les deux registres. Par ailleurs, le bloc "**Fonction d'inversion (Num_bit2)**" effectue une deuxième inversion sur la donnée modifiée précédemment. Cette fois-ci, la cible à inverser est déterminée en fonction de la valeur affectée à l'entrée primaire "Num_bit2".
- La deuxième fonction "**Fonction d'affectation finale**" est dédiée à la définition de la valeur finale au niveau de l'entrée des registres. Cette valeur est déterminée par l'entrée de contrôle "En_inj". Si cette dernière est égale à zéro, les registres seront assignés par les données des entrées primaires "DataIn1" et "DataIn2" (fonctionnement nominal). Par contre si la valeur de l'entrée de contrôle "En_inj" n'est pas égale à zéro, au moins l'un des bits des deux registres sera inversé. "En_inj" détermine le nombre d'inversions (dans ce cas, le nombre maximal est égal à 2).

4.3. Gestion hiérarchique

Pour un circuit modélisé par une description hiérarchique et candidat à une injection de fautes, nous devons définir une méthode de propagation et de gestion des entrées de contrôle de l'injection qui soit optimale et automatisable. Les modifications doivent permettre au concepteur de cibler des éléments se trouvant dans des blocs hétérogènes et cela quelque soit la profondeur de la hiérarchie.

L'architecture globale d'une hiérarchie instrumentée est illustrée dans la Figure 4-14. Pendant la génération du mutant, toutes les modifications fonctionnelles dans les blocs décrits en VHDL comportemental, ainsi que la propagation des signaux supplémentaires dans la hiérarchie sont réalisées automatiquement. N'importe quel point mémoire dans la hiérarchie peut être modifié pour injecter des inversions de bits uniques ou multiples, ou encore des transitions erronées si la cible est définie par des valeurs symboliques. Pour cela, les signaux de commande d'injection sont propagés aux entrées des différents blocs cibles du circuit et, lorsque nécessaire, des signaux internes (spécifiés par le concepteur comme étant des points à observer) sont également propagés à des sorties externes additionnelles. Le nombre des signaux ajoutés pour localiser les bits à inverser est défini en fonction de la multiplicité spatiale maximale spécifiée par le concepteur. Ces signaux sont décodés dans la hiérarchie du circuit.

Grâce au signal de sélection de bloc "Sel_comp", l'injection de fautes dans un sous-bloc donné peut être contrôlée indépendamment des autres sous-blocs qui composent la hiérarchie du circuit (chaque registre étant situé dans un sous-bloc donné). La réduction du surcoût global implique la réduction des surcoûts locaux (liés aux blocs internes ciblés par l'injection). Afin de réduire au maximum les surcoûts locaux, la taille des signaux de contrôle ajoutés au niveau d'un bloc interne cible sera optimisée en fonction du nombre total des bits cibles inclus dans ce bloc.

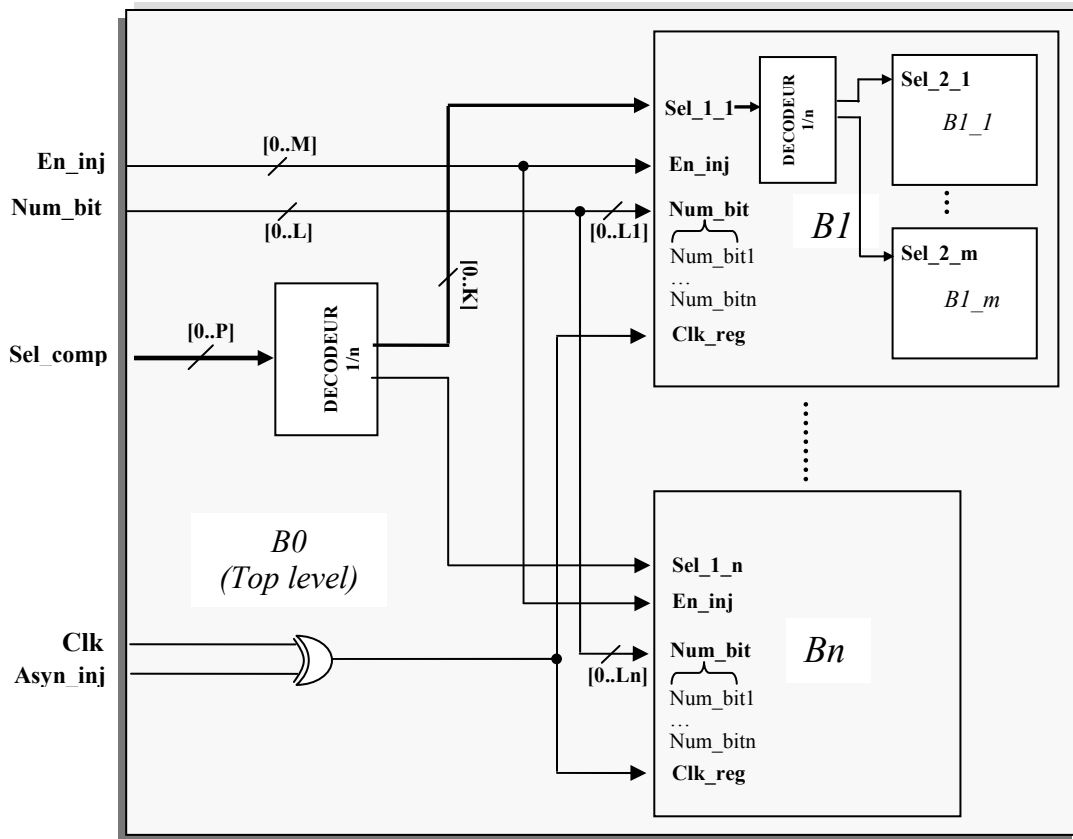


Figure 4-14: Description hiérarchique modifiée pour l'injection de fautes de type MBF.

Le principe général de la méthode consiste à modifier les blocs cibles inclus dans la hiérarchie et ensuite, à propager de manière ascendante toute information utile qui permettra au concepteur d'atteindre l'élément cible et ainsi de réaliser l'injection de fautes.

Nous commençons, tout d'abord, par agir sur les feuilles de la hiérarchie, c'est-à-dire appliquer le processus de mutation sur les composants qui n'incluent pas de composants internes. Pour ces composants, les mutants et les entrées de contrôle seront insérés et optimisés en ne tenant compte d'aucun paramètre extérieur au composant. Ensuite, les mêmes entrées de contrôle seront ajoutées au niveau supérieur. Si le composant du niveau supérieur inclut plus d'un composant, une entrée supplémentaire lui sera ajoutée ainsi qu'à tous ses blocs internes cibles (cela correspond à tous les signaux de type "Sel_i_i" dans la Figure 4-14). Lors d'une expérience d'injection, c'est l'entrée "Sel_comp" qui, une fois adressée, permet de désigner parmi les composants cibles celui qui sera effectivement la cible de l'injection.

Par ailleurs, la fonction qui réalise le contrôle externe du signal d'horloge sera modélisée au niveau hiérarchique le plus haut. L'entrée primaire "**Asyn_inj**" sera combinée avec le signal d'horloge "Clk". Le signal généré "Clk_reg" sera propagé dans tous les blocs internes en remplaçant dans ces blocs l'entrée d'horloge "Clk" par "Clk_reg" (Figure 4-14).

➤ **En VHDL :**

Nous allons spécifier dans la suite les différentes modifications VHDL pour obtenir une version hiérarchique mutée. La Figure 4-15 illustre l'insertion des différentes entrées requises pour l'injection dans la partie entité.

Concernant la définition de la taille des sélecteur de bits "**Num_bit_i**" ajoutés au composant du niveau supérieur, la borne supérieure sera égale à la valeur maximale des bornes supérieures des entrées "Num_bit_i" insérées aux différents composants cibles internes. Le signal d'entrée "**En_inj**" est utilisé pour définir la multiplicité spatiale. Cette information est propagée dans tous les blocs concernés par l'injection. Pour tous ces blocs, le signal "En_inj" aura la même taille, qui correspond à la multiplicité spatiale maximale définie par le concepteur.

```
entity Top_Level_Entity is
port
  (-- inputs
  Clk, Reset : in std_logic ;
  ...
  -- Ports d'entrée/sortie nécessaires pour l'injection
  Sel_comp : in std_logic_vector(P downto 0);      -- Sélection du composant cible
  En_inj : in std_logic_vector(M downto 0);         -- Définition de la multiplicité spatiale
  En_asyn : in std_logic;                          -- contrôle asynchrone de l'injection
  Num_bit1 : in integer range 0 to m-1;             -- Numéro du 1er bit à inverser
  Num_bit2 : in integer range 0 to m-1;             -- Numéro du 2ème bit à inverser
  ...
  Num_bitn : in integer range 0 to m-1              -- Numéro du nème bit à inverser
);
end Top_Level_Entity;
```

Figure 4-15: Insertion des signaux de contrôle dans les différents blocs de la hiérarchie

- **Processus de gestion des composants internes**

Comme nous l'avons déjà mentionné, l'entrée "Sel_comp" permet à l'instant d'injection de mettre un seul des composants cibles en mode d'injection (les autres seront en mode de mémorisation). En effet, l'injection multiple reste limitée à un seul composant dont les points cibles sont choisis par le biais des entrées de sélection "Num_bit_i". Cette stratégie est modélisée par une fonction décrite en VHDL et illustrée par les Figures 4-16-(a) et 4-16-(b) qui représentent deux cas différents. Le processus de la Figure 4-16-(a) est inséré dans toutes les descriptions de profondeur hiérarchique égale à 1 (exemple du composant B1 dans la Figure 4-14) alors que celui de la Figure 4-16-(b) sera inséré dans des descriptions de profondeur hiérarchique supérieur à 1 (exemple du composant B0 dans la Figure 4-14).

Dans les deux cas, le processus de sélection de bloc inclut une instruction "case" contrôlée par un sélecteur de composant. Il modélise un décodeur 1 parmi n, avec en plus deux cas qui correspondent à la mise à 0 ou à 1 de toutes les sorties du décodeur. Chaque sortie du décodeur est connectée à une entrée de sélection de bloc d'un composant cible. Lorsque cette entrée est mise à 1, le composant correspondant est en mode de fonctionnement normal (si l'entrée "En_inj" est nulle) ou en mode d'injection (si "En_inj" est différent de zéro). Dans le cas contraire (c-à-d le cas où l'entrée est mise à 0), le composant correspondant est en mode de mémorisation ; ceci est possible grâce à l'insertion des instructions conditionnelles contrôlées par le signal de sélection de bloc dans tous les processus séquentiels modélisant les éléments cibles (illustrée par le signal "Sel_x" dans la Figure 4-10 de la section 4.2.3).

```

process (Sel_1)
begin
  case Sel_1 is
  when "@ du composant B1" =>
    Sel_2_1 <= '1';
    Sel_2_2 <= '0';
    ...
    Sel_2_n <= '0';
  when "@ du composant B2" =>
    Sel_2_1 <= '0';
    Sel_2_2 <= '1';
    ...
    Sel_2_n <= '0';
  ...
  when "@ du composant Bn" =>
    Sel_2_1 <= '0';
    Sel_2_2 <= '0';
    ...
    Sel_2_n <= '1';
  when "tous les bits à 0" =>
    Sel_2_1 <= '0';
    Sel_2_2 <= '0';
    ...
    Sel_2_n <= '0';
  when "tous les bits à 1" =>
    Sel_2_1 <= '1';
    Sel_2_2 <= '1';
    ...
    Sel_2_n <= '1';
  when others => null;
end case;
end process;

```

(a)

```

process (Sel_comp)
begin
  case Sel_comp is
  when "@ du bloc B1_1" => Sel_1_1 <= "@ B1_1";
    Sel_1_2 <= '0';
    ...
    Sel_1_n <= '0';
  ...
  when "@ du bloc B1_m" => Sel_1_1 <= "@ B1_m";
    Sel_1_2 <= '0';
    ...
    Sel_1_n <= '0';
  when "@ du bloc B2" => Sel_1_1 <= "tous à 0";
    Sel_1_2 <= '1';
    ...
    Sel_1_n <= '0';
  when "@ du bloc Bn" => Sel_1_1 <= '0';
    Sel_1_2 <= '0';
    ...
    Sel_1_n <= '1';
  when "tous les bits à 0" => Sel_1_1 <= 'tous à 0';
    Sel_1_2 <= '0';
    ...
    Sel_1_n <= '0';
  when "tous les bits à 1" => Sel_1_1 <= 'tous à 1';
    Sel_1_2 <= '1';
    ...
    Sel_1_n <= '1';
  when others => null;
  end case;
end process;

```

(b)

Figure 4-16: Fonctions utilisées pour désactiver les composants non concernés par l'injection asynchrone. Dans (a) la profondeur hiérarchique est égale à 1 alors que dans (b) elle est supérieure à 1.

Dans le cas de la fonction illustrée par la Figure 4-16-(a), toutes les sorties sont codées sur un seul bit car elles sont connectées à des composants qui n'incluent pas de composants internes. Dans le cas contraire, les composants internes (exemple : les blocs B1_1 ... B1_m de

la Figure 4-14) doivent être adressés un par un comme cela est indiqué dans la Figure 4-16-(b). Par conséquent, la taille du signal "Sel_comp" dans le "Top level" sera calculée en fonction du nombre total des composants cibles internes se trouvant à n'importe quelle profondeur hiérarchique (en ajoutant deux adresses pour la mise à 0 et à 1 de toutes les sorties). Par convention, tous les bits du sélecteur de composant doivent être à 0 pour la mise à 0 des sorties et à 1 pour la mise à 1.

Le protocole d'injection, présenté dans la section 3.4, a été généralisé pour une description hiérarchique. Dans le nouveau protocole, illustré par la Figure 4-17, nous représentons le chronogramme du signal "Sel_comp" en plus des deux signaux de contrôle d'injection "Asyn_inj" et "En_inj".

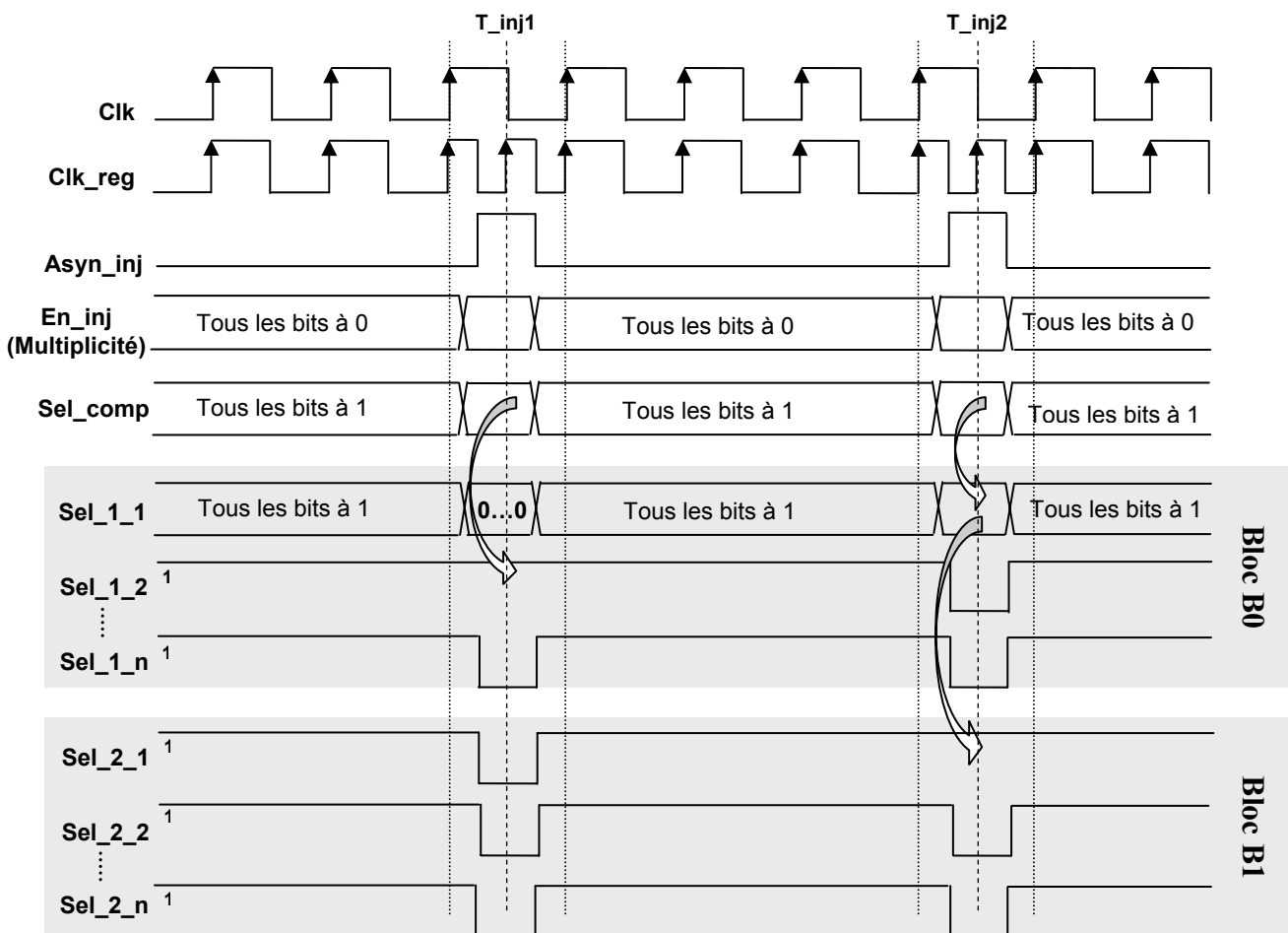


Figure 4-17: Protocole d'injection

La Figure 4-17 montre l'évolution des signaux générés en sortie des deux décodeurs des composants B0 et B1. Ils sont contrôlés par le signal "Sel_comp". Pour garantir un fonctionnement nominal du circuit, il faut maintenir les bits de l'entrée "En_inj" à 0, tandis que les bits du sélecteur "Sel_comp" doivent être mis à 1. Il est possible de générer de nouveaux fronts d'horloge par le biais du signal "Asyn_inj". Nous appliquons alors deux injections au milieu de deux cycles d'exécution (instants "T_inj1" et "T_inj2"). Cependant, pour le bon

déroulement d'une injection asynchrone, nous devons respecter la contrainte suivante : l'activation des signaux "En_inj" et "Sel_comp" doit s'effectuer entre les deux fronts d'horloge d'un cycle d'injection (voir Figure 4-17).

Au cycle d'injection "T_inj1", le signal "Sel_comp" cible le bloc B2. Par conséquent l'entrée de sélection "Sel_1_2" est mise à 1 et tous les autres signaux sont mis à 0. Pour le deuxième cycle d'injection "T_inj2", l'adresse du signal "Sel_comp" désigne pour l'injection le bloc B1_1. En effet, l'adresse de ce dernier est d'abord affectée à l'entrée "Sel_1_1". Le décodeur 1 parmi n maintient le signal de sélection "Sel_2_1" à 1 et met tous les autres blocs en mode de mémorisation.

• Exemple

Nous présentons dans la Figure 4-18 le schéma bloc d'un circuit modélisé par une description hiérarchique de profondeur deux. Le circuit est instrumenté pour implémenter le modèle de faute MBF. L'entrée de sélection "Sel_comp" sera utilisée pour adresser les trois bloc : B1_1, B1_2 et B2. Cependant, elle est implémentée sur trois bits afin de prendre en compte la mise à 0 et à 1 de tous les sélecteurs de blocs internes. Les décodeurs des blocs B0 et B1 sont décrits en VHDL par les deux processus de la Figure 4-19.

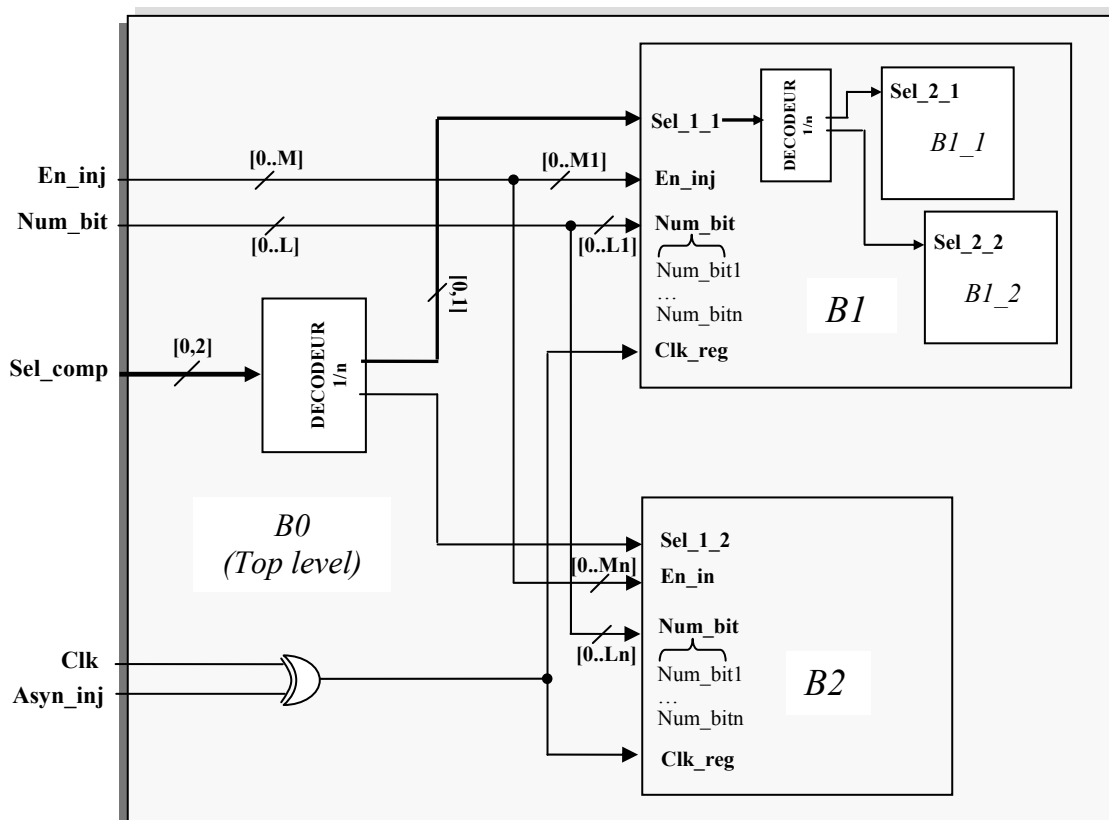


Figure 4-18: Représentation structurelle d'une description hiérarchique de profondeur 2 modifiée pour l'injection de fautes type MBF.

```

process (Sel_1_1)
begin
  case Sel_1_1 is
    when "01" =>
      Sel_2_1 <= '1';
      Sel_2_2 <= '0';
    when "10" =>
      Sel_2_1 <= '0';
      Sel_2_2 <= '1';
    when "11" =>
      Sel_2_1 <= '1';
      Sel_2_2 <= '1';
    when "00" =>
      Sel_2_1 <= '0';
      Sel_2_2 <= '0';
    when others => null;
  end case;
end process;

```

(a)

```

process (Sel_comp)
begin
  case Sel_comp is
    when "001" =>
      Sel_1 <= "01";
      Sel_2 <= '0';
    when "010" =>
      Sel_1 <= "10";
      Sel_2 <= '0';
    when "011" =>
      Sel_1 <= "00";
      Sel_2 <= '1';
    when "111" =>
      Sel_1 <= "11";
      Sel_2 <= '1';
    when "000" =>
      Sel_1 <= "00";
      Sel_2 <= '0';
    when others => null;
  end case;
end process;

```

(b)

Figure 4-19: Fonctions utilisées pour gérer l'injection asynchrone. Les processus décrits dans (a) et (b) seront respectivement insérés dans les descriptions des composants B0 et B1.

4.4. Compromis entre la généralité et la complexité

Le processus de mutation présenté dans la section 4.2.3 permet d'effectuer des injections avec une seule limitation : le nombre maximum d'inversions simultanées au niveau d'un sous-bloc donné. Les inversions de bits multiples peuvent se localiser n'importe où dans les éléments cibles d'un bloc donné, et il n'y a également aucune limitation par rapport à la multiplicité temporelle puisque l'injection est contrôlée par des entrées primaires ajoutées au circuit. Le prix à payer pour satisfaire cette généralité peut ne pas être acceptable dans certains cas, particulièrement quand un prototypage matériel est utilisé pour exécuter les expériences. Par conséquent, nous avons implémenté un autre type de processus de mutation présenté dans la section 4.2.2, en supposant que tous les bits modifiés simultanément sont inclus dans un seul registre. Dans ce cas, il n'y a aucun besoin de générer le registre virtuel mentionné précédemment.

Par ailleurs, il serait possible d'implémenter également une mutation qui tiendrait compte d'un sous-ensemble de registres donné, mais dans la pratique, il serait beaucoup plus difficile de spécifier ces sous-ensembles lors de la définition de la campagne. Finalement, l'hypothèse serait soit de limiter les injections à un seul registre pour chaque expérience (avec une multiplicité d'erreur quelconque) ou d'utiliser les mutants généraux en divisant potentiellement la campagne en plusieurs sous-campagnes, chacune d'elle étant associée à un sous-ensemble de cibles d'injection.

Il faut noter que l'approche présentée peut facilement être adaptée pour permettre d'injecter des fautes multiples simultanément sur l'ensemble du circuit. Il suffirait, en première approche,

de supprimer la hiérarchie des décodeurs de sélection des sous-blocs. Ceci entraînerait un fort accroissement du nombre d'entrées primaires additionnelles, nuisible pour une implémentation prototypée. Par ailleurs, la qualité globale de l'analyse faite sur le circuit n'en serait pas forcément améliorée si les différents blocs du circuit permettent un niveau minimum de confinement des erreurs.

4.5. Conclusion

Nous avons présenté dans ce chapitre plusieurs raisons qui justifient la prise en compte des inversions de bits multiples (en plus des modèles de fautes habituellement considérés dans la littérature) lors de l'analyse de sûreté effectuée très tôt dans le flot de conception. La génération automatique de mutants VHDL permettant d'effectuer de telles injections de fautes a été présentée. Les mutants utilisés permettent d'injecter plusieurs types de modèles de fautes selon des contraintes liées à l'application et selon le niveau de description des éléments de mémorisation. Les modèles de fautes hétérogènes incluent les "inversions de bits" uniques et multiples ainsi que les transitions erronées. Un mutant donné permet au concepteur, à partir d'une interface d'injection unique et cohérente, de choisir le bon modèle de fautes associé à chaque élément de mémorisation.

La possibilité de faire des compromis entre la généralité du modèle de fautes et la complexité des mutants générés a été discutée. Nous présentons des résultats expérimentaux et nous discutons de l'impact de l'injection de MBF dans le chapitre suivant.

Chapitre 5.

Expériences d'injection

5.1. Injections multi-niveaux

Dans cette section, nous allons évaluer l'efficacité d'une analyse de sûreté effectuée très tôt dans le flot de conception. L'idée proposée consiste à analyser la corrélation entre des résultats d'injection obtenus à différents niveaux de spécification du circuit. Les véhicules de test choisis sont un coprocesseur arithmétique et un processeur Fuzzy.

5.1.1. Co-processeur de multiplication entière

5.1.1.1. Descriptions

Le premier véhicule de test est un noyau arithmétique de coprocesseur simple exécutant des multiplications non signées de 16 bits en utilisant un algorithme séquentiel. Des expériences d'injection ont été effectuées sur deux descriptions très différentes du même circuit. Les deux descriptions sont fonctionnellement équivalentes (mêmes comportements externes) mais sont très différentes en terme de structure interne.

Le niveau de description le plus élevé, appelé "architecture", correspond à une modélisation RTL à plat d'un graphe de contrôle. Cette description est synthétisable mais n'est pas optimisée en terme de structure matérielle. Les spécifications des opérations liées à l'état "multin" sont illustrées dans la Figure 5-1. Nous constatons par exemple que le compteur interne de boucle est ici spécifié en utilisant deux variables (counter, ncounter) et une incrémentation.

```
when multin =>
  Valid_temp <= '1';
  if (operandB(counter)='1') then
    nstate <= addit;
    naccumulation <= accumulation + operandA;
    noperandA <= operandA;
    ncounter <= counter;
  else nstate <= shifting;
    noperandA <= operandA + operandA;
    ncounter <= counter + 1;
    naccumulation <= accumulation;
  end if;
```

Figure 5-1: Spécification de l'état "multin" de la description "architecture".

Le deuxième niveau de description, qui s'appellera "micro-architecture", est aussi une description RTL synthétisable. Cependant, la structure matérielle est ici beaucoup plus optimisée. La description est hiérarchique, avec séparation de la partie contrôle et de la partie opérative comme illustré dans la Figure 5-2. La structure niveau bloc de la partie opérative est décrite en VHDL comportemental. Les différences avec la description "architecture" sont illustrées dans la Figure 5-3. La Figure 5-3(a) montre la spécification de l'état "multin" dans la partie contrôle qui est modélisée par une machine d'état fini. Tandis que le compteur de boucle de la partie opérative est décrit dans la Figure 5-3(b). Dans ce cas, le compteur est implémenté comme un registre à décalage, ce qui est très différent de la description "architecture" mais correspond à la même fonctionnalité dans le circuit.

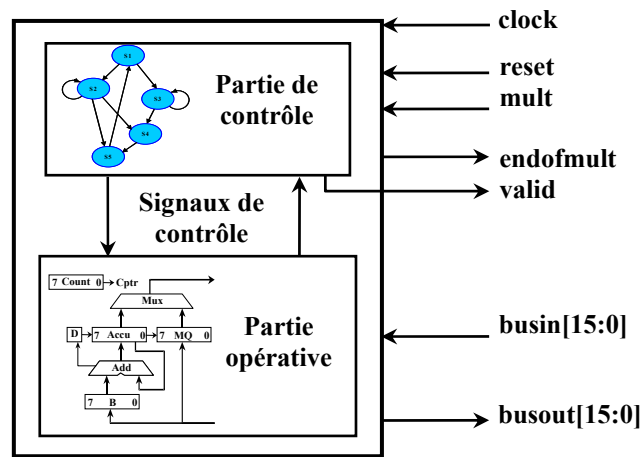


Figure 5-2: Structure hiérarchique du multiplieur.

```

when multin =>
  Valid_temp <= '1';
  SelMQFF c= '0';
  InitG <= '1';
  SelD <= '1';
  SelB <= '0';
  SelBusOut_temp <= '0';
  if (Mq0='1') then nstate <= addit;
    SelACCF <= '1';
    InitD= '1';
    SelACCMux <= '0';
    SelMQMux <= '0';
    SelCPTR <= '0';
  else nstate <= shifting;
    SelACCF <= '0';
    InitD <= '0';
    SelACCMux <= '1';
    SelMQMux <= '1';
    SelCPTR <= '1';
  end if;

```

(a)

```

if (SelCPTR='1') then
  nCPTR(nbbits-1 downto 0) <= '1' & CPTR(nbbits-1 downto 1);
elsif (InitCPTR='0') then
  nCPTR <= "0000000000000000";
else nCPTR <= CPTR;
end if; -- description du compteur

```

(b)

Figure 5-3: Une partie de la description "micro-architecture" avec (a) la partie contrôle et (b) la partie opérative.

La grande différence entre ces deux descriptions est également illustrée par les résultats de synthèse montrés dans le Tableau 5-1. La "micro-architecture" est réellement beaucoup plus raffinée que la description "architecture" et optimise considérablement la surface et les performances.

Tableau 5-1: Résultat de Synthèse pour la spécification "architecture" et "micro-architecture" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).

	Architecture	μ -Architecture
Nombre d'instances	332	214
Surface (sqmils)	355	231
Fréquence Max. (MHz)	54	96

Des expérimentations ont été réalisées sur ce coprocesseur arithmétique afin de valider l'efficacité d'une analyse de sûreté effectuée très tôt dans le flot de conception. Le but de ces expériences est d'évaluer la corrélation entre les chemins de propagation d'erreur identifiés sur différentes descriptions du même circuit. Une analyse très tôt d'une description de haut niveau d'un circuit est significative seulement si des chemins de propagation semblables sont observés dans le cas d'une analyse de descriptions plus raffinées du même circuit.

5.1.1.2. Injection de SEU

Lors des expériences d'injection rapportées dans cette section, nous nous sommes focalisés sur des erreurs se produisant dans la partie contrôle du circuit, et plus spécifiquement dans les registres d'états.

Nous avons effectué des injections de fautes "inversions de bit" et "transitions erronées" sur les deux descriptions modélisant le multiplieur, en utilisant le processus de mutation décrit dans les sections 3.2 et 3.3. Dans le cas des injections de type "inversion de bit", un codage optimisé sur trois bits a été utilisé. Pour les quatre cas, nous allons générer le type de modèle comportemental décrit dans la section 2.3.2 et cela en tenant compte :

- Soit uniquement de l'observation des signaux externes, soit de l'observation des signaux externes et de certains signaux internes. **Les signaux externes** observés sont le bus de données "busout", le signal "endofmult" indiquant la fin du calcul et le signal "valid" pour indiquer la disponibilité d'un résultat sur le bus de sortie. **Les signaux internes** observés sont le registre d'états et le compteur de boucle. Lors de la génération du modèle comportemental, chaque variable ou bus "multi-bit" observé est considéré comme une boîte noire, c.-à-d. soit correct soit incorrect, sans analyser la valeur incorrecte.
- Soit d'une inversion de bit unique qui provoque seulement des transitions erronées vers des **états valides**, soit d'une inversion de bit unique qui peut conduire à des **états invalides** (il y

a un seul état invalide pour le véhicule de test étudié qui est codé 000 après le codage des états).

- Soit d'une **analyse complète** de la propagation d'erreur, soit de la **définition d'états terminaux**. Dans le premier cas, les propagations sont analysées jusqu'à la fin de chaque expérience. Dans le deuxième cas, l'analyse de chaque expérience est arrêtée si une des conditions spécifiées par le concepteur est vérifiée. Les conditions définies pour ces expériences sont :

- (1) **Détection** d'erreur si les signaux de contrôle "mult" et "valid" sont tous les deux au niveau actif (mult=1 et valid=0) puisqu'une telle condition ne devrait pas se produire en mode normal,
- (2) **Défaillance** si "endofmult" est incorrect et au niveau actif,
- (3) **Défaillance** si "busout" est incorrect et "valid" est au niveau actif,
- (4) **Défaillance** si "valid" est incorrect.

Il a été généré donc un total de 24 modèles comportementaux qui montrent les propagations d'erreur dans le circuit étudié. Toutes les transitions erronées possibles ou toutes les inversions uniques possibles ont été injectées à chaque cycle d'horloge pendant l'exécution du programme de test. Dans ces expériences, aucun mécanisme particulier de détection n'a été implémenté dans le circuit. Les Tableaux 5-2 et 5-3 résument les états et transitions des 12 graphes obtenus sans états terminaux, sans inclure les deux états de base "Correct" et "Latent". Ceci permet de comparer les résultats obtenus sur une base qualitative.

Tableau 5-2: Caractéristiques des modèles comportementaux générés (observation des signaux externes seulement, aucun état terminal).

Injection	Etats		Transitions		Transitions communes
	Archi	μ -Archi	Archi	μ -Archi	
Trans. Err.	8 (100%)	8 (100%)	49 (67%)	49 (67%)	42 (86%)
Inv. Bit (états valides)	8 (100%)	8 (100%)	40 (55%)	48 (66%)	38 (79%)
Inv. Bit (état 000 inclut)	8 (100%)	8 (100%)	40 (55%)	48 (66%)	38 (79%)

Tableau 5-3: Caractéristiques des modèles comportementaux générés (observation des signaux externes et internes, aucun état terminal).

Injection	Etats		Transitions		Transitions communes
	Archi	μ -Archi	Archi	μ -Archi	
Trans. Err.	22 (69%)	20 (63%)	152 (14%)	138 (13%)	107 (78%)
Inv. Bit (états valides)	21 (66%)	20 (63%)	125 (12%)	125 (12%)	86 (69%)
Inv. Bit (état 000 inclut)	21 (66%)	20 (63%)	125 (12%)	125 (12%)	86 (69%)

Le Tableau 5-2 résume les résultats obtenus lorsque seuls les signaux externes sont observés. Un total de 3 signaux ou variables est observé ce qui correspond à un maximum de 7

configurations d'erreur possible. En conditions d'expérience, le graphe généré peut faire apparaître à la suite de l'état "Latent", un maximum de 8 états (les 7 "états d'erreur" et l'état "sans erreur"). Le nombre maximum des transitions qui peuvent apparaître dans le graphe, après que l'état "Latent" ne soit atteint, est de 73 : une boucle sur "Latent", 8 transitions aux successeurs potentiels, et 64 transitions possibles entre les 8 successeurs possibles. Comme montré dans le tableau, tous les états possibles ont été atteints, peu importe le niveau de description ou le type d'erreur injectée. Cependant, parmi les 73 transitions qui peuvent potentiellement être activées pendant les propagations d'erreur, seulement 55% à 67% sont enregistrés réellement. Comme mentionné en introduction, ceci démontre l'efficacité d'une telle analyse dans l'identification des événements les plus critiques et donc des points optimaux où des mécanismes de protection devraient être insérés pour le durcissement de la conception. Cela est plus évident dans le cas où les états d'erreur internes et les valeurs du compteur sont pris en compte ; les résultats sont montrés dans le Tableau 5-3. Le graphe généré peut faire apparaître jusqu'à 32 états et 1057 transitions. Il a été observé réellement moins de 70% des configurations d'erreur possibles et moins de 15% des transitions possibles ont été enregistrées entre les états correspondants.

La comparaison des résultats obtenus pour les deux descriptions et les deux types d'injection montre par ailleurs une bonne cohérence. En particulier, le nombre de transitions communes aux graphes obtenus pour les deux niveaux de description et un type d'injection donné (c.-à-d. le nombre des propagations d'erreur identiques dans les deux descriptions) est supérieur à 69% et peut atteindre plus de 80% (cf. dernière colonne du tableau). Ceci démontre qu'une campagne d'injection réalisée très tôt dans le processus de conception, sur une description encore très éloignée de l'implémentation finale, peut donner des informations très utiles sur les caractéristiques de sûreté d'un circuit.

Les graphes les plus complexes ont été obtenus en injectant des transitions erronées dans la description "architecture", ce qui confirme les commentaires de la section 3.2. Cependant, considérer le plus mauvais cas à une étape préliminaire de la conception peut éviter ou réduire des itérations très coûteuses et très longues dans le processus de conception (et ainsi économiser beaucoup de temps et d'effort).

Par ailleurs, les injections de fautes de type inversion de bit qui aboutissent à l'état invalide (code 000) n'ont pas modifié la structure des graphes générés ; seules les probabilités associées à quelques arcs ont été affectées.

Comme illustré dans le Tableau 5-4, les graphes obtenus après définition des états terminaux sont plus simples, puisque dans ce cas, les propagations ne sont pas analysées après l'apparition d'une défaillance. Cependant, le pourcentage des transitions communes en analysant les deux niveaux de description "architecture" et "micro-architecture" continue d'augmenter. Ceci prouve qu'une partie des différences observées dans les résultats précédents correspondent à des propagations qui ne sont pas vraiment significatives pour l'application puisqu'elles se produisent après que l'application ait atteint un état inacceptable.

Tableau 5-4: Caractéristiques des modèles comportementaux générés avec spécification des états terminaux.

Signaux observés	Injection	Etats		Transitions		Transitions communes
		Archi	μ -Archi	Archi	μ -Archi	
Externes	Trans. Err.	7 (64%)	7 (64%)	25 (26%)	27 (28%)	25 (93%)
	Inv. Bit	7 (64%)	7 (64%)	24 (25%)	27 (28%)	24 (88%)
Externes et internes	Trans. Err.	16 (46%)	17 (49%)	74 (6%)	71 (6%)	54 (76%)
	Inv. Bit	16 (46%)	16 (46%)	64 (6%)	57 (5%)	42 (74%)

Finalement, le Tableau 5-5 présente le nombre de transitions communes aux graphes obtenus au plus haut niveau (injection de transitions erronées dans la description "architecture") et au plus bas niveau (injection de type inversion de bit dans la description "micro-architecture"). Selon la définition de la campagne (avec ou sans les signaux internes à observer et avec ou sans la définition des états terminaux), entre 77% et 93% des propagations d'erreur observées à bas niveau peuvent déjà être observé à haut niveau.

Tableau 5-5: Comparaison des résultats d'analyse entre le plus haut et le plus bas niveau - nombre de transitions communes des modèles comportementaux obtenus après injection des transitions erronées dans la description "architecture" et des inversions de bit dans la description "micro-architecture"

Signaux observés	Pas d'état terminal	Avec état terminal
Externes	43 (90%)	25 (93%)
Externes et internes	100 (80%)	44 (77%)

D'autres expériences ont été effectuées sur une troisième modélisation du coprocesseur arithmétique. Celle-ci est inspirée de la description "architecture" et correspond aussi à une modélisation RTL à plat d'un graphe de contrôle. Mais ce dernier est plus optimisé avec un nombre d'états limité à cinq au lieu de sept pour la description "architecture". Par ailleurs, le compteur est implémenté comme un registre à décalage. Cette description, qui sera appelée "architecture 2", est fonctionnellement équivalente aux deux autres mais différente en terme de structure interne. Les résultats de synthèse sont illustrés par le Tableau 5-6. La description "architecture 2" est plus optimisée que la description "architecture" en termes de surface et de performances.

Tableau 5-6: Résultat de Synthèse pour la spécification "architecture 2" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).

	Nombre d'instances	Surface (sqmils)	Fréquence Max. (MHz)
Architecture 2	220	263	97

Les Tableaux 5-7 et 5-8 résument respectivement les résultats obtenus lorsque sont observés seulement les signaux externes ou les signaux internes et externes. Pour le cas des inversions de bits, il y a en tout trois états invalides 000, 110 et 111. Le comportement en

présence de fautes du nouveau modèle "architecture 2" reste proche des deux descriptions "architecture" et "micro-architecture". En effet, le nombre de transitions communes entre le graphe obtenu pour la description "architecture 2" et les autres descriptions est supérieur à 56% et peut atteindre 75% (cf. dernière colonne du tableau). Compte tenu du niveau de raffinement de la description "architecture 2", il est logique que les résultats soient plus proches de ceux de la description "micro-architecture", même si la spécification est de plus haut niveau.

Tableau 5-7: Caractéristiques des modèles comportementaux générés (observation des signaux externes seulement, aucun état terminal).

Injection	Etats	Transitions	Transitions communes avec	
	Archi2 (5 états)	Archi2 (5 états)	Archi (7 états)	μ -Archi
Trans. Err.	8 (100%)	51 (70%)	32(63%)	38(75%)
Inv. Bit (états valides)	8 (100%)	48 (66%)	27(56%)	35(73%)
Inv. Bit (états 000, 110 et 111 inclus)	8 (100%)	48 (66%)	27(56%)	35(73%)

Tableau 5-8: Caractéristiques des modèles comportementaux générés (observation des signaux externes et internes, aucun état terminal).

Injection	Etats	Transitions	Transitions communes avec	
	Archi2 (5 états)	Archi2 (5 états)	Archi (7 états)	μ -Archi
Trans. Err.	25 (78%)	130 (12%)	70(54%)	66(51%)
Inv. Bit (états valides)	21 (66%)	104 (10%)	52(42%)	51(41%)
Inv. Bit (états 000, 110 et 111 inclus)	21 (66%)	104 (10%)	52(42%)	51(41%)

Une comparaison des résultats d'analyse entre le plus haut et le plus bas niveau a été effectuée. Le nombre de transitions communes des modèles comportementaux obtenus après injection des transitions erronées dans la description "architecture 2" (5 états) et injection des inversions de bits dans la description "micro-architecture" est égal à 63 (48%), sans définition d'états terminaux et avec observation des signaux externes et internes.

5.1.2. Processeur Fuzzy

Nous avons exécuté des expériences semblables sur deux descriptions d'un processeur de calcul en logique floue.

5.1.2.1. Description

La partie contrôle du processeur Fuzzy a été modélisée à deux niveaux d'abstraction : le premier niveau de description (qui sera appelé "micro-contrôleur") correspond à la description initiale du contrôleur (voir annexe A.). Le deuxième niveau (qui sera appelé "contrôleur") correspond à une description moins raffinée (l'exemple est décrit dans l'annexe B.). Les deux

descriptions correspondent au même comportement externe mais l'architecture interne est différente.

A présent, nous allons montrer comment, à partir du modèle initial du contrôleur, nous avons obtenu une description plus abstraite.

Tableau 5-9: Les différents processus inclus dans la description "micro-contrôleur".

Processus	P_counter	P_pipeline	P_counter2	P_accumulation
Sig. de contrôle	clk_reg = 0	clk = 1	clk = 0	clk = 1
Condition	counter	counter	int_pipe	counter
Registre	counter	int_pipe	counter2	accumulation

La description "micro-contrôleur" comporte plusieurs processus utilisés pour implémenter les différents registres du circuit. Certaines caractéristiques de ces processus sont résumées dans le Tableau 5-9. Ce dernier montre que l'assignation des signaux "int_pipe", "counter2" et "accumulation" dépend directement ou indirectement de la valeur du signal "counter". Par exemple, l'affectation du signal "counter2" dépend du résultat de l'expression conditionnelle du signal "int_pipe" qui, de même, dépend de la valeur du signal "counter". Par conséquent, le signal "counter" peut être considéré comme un registre d'états où pour chaque état, on fait correspondre les expressions et affectations associées aux différents processus de la description "micro-contrôleur". Le signal "counter" sera donc décrit dans un processus séquentiel sensible au front descendant de l'horloge "clk_reg" (voir annexe B.), tandis que la deuxième partie du FSM, combinatoire, est utilisée pour déterminer les transitions entre les états et la valeur des signaux "naccumulation", "nint_loop", "nint_pipe" et "ncounter2" (spécifiée sur les états). Ces signaux correspondent respectivement aux valeurs d'entrée des registres "accumulation", "int_loop", "int_pipe" et "counter2" qui seront affectées dans le processus séquentiel.

Par ailleurs, dans la description "micro-contrôleur", les processus "P_pipeline" et "P_accumulation" sont sensibles au front montant de l'horloge "clk" (contrairement aux processus "P_counter" et "P_counter2"). Pour que cet aspect soit modélisé dans la description "contrôleur", il suffit d'ajouter le processus "P_acc_int" réalisant l'affectation des registres "accumulation" et "int_loop" au front montant de l'horloge.

La différence entre les deux descriptions est illustrée aussi par les résultats de synthèse (Tableau 5-10). La description initiale est en réalité beaucoup plus raffinée en termes de surface et de performance que la description "Contrôleur".

Tableau 5-10: Résultat de Synthèse pour la spécification "contrôleur" et "micro-contrôleur" (en utilisant l'outil de synthèse Leonardo et la bibliothèque de cellules standards "cub" d'AMS).

	Contrôleur	μ-Contrôleur
Nombre d'instances	41	32
Surface (sqmils)	38	25
Fréquence Max. (MHz)	281,1	314,9

5.1.2.2. Injection de SEU

Des expériences d'injection de fautes simples ont ciblé au niveau RTL le registre "counter" de la description "micro-contrôleur" et le registre d'états "state" de la description "contrôleur". Le signal "counter" est un compteur de type entier défini dans l'intervalle [0..5]. Il met 6 cycles d'horloge pour passer de 0 à 5. Alors que dans la description "Contrôleur", c'est le registre d'états qui joue le rôle du compteur ; il prend la valeur "init", passe à l'état "loopState", boucle dans cet état pendant 4 cycles d'horloge et ensuite passe à l'état "endState" (ce qui fait un total de 6 cycles d'horloge).

Le Tableau 5-11 et le Tableau 5-12 résument respectivement les valeurs des registres "counter" et "state" obtenues suite à des inversions de bit uniques contrôlées par le signal d'entrée "Sel".

Tableau 5-11: Résultats d'injection de SEU sur le registre "counter".

counter Sel	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)
'01'	1 (001)	0 (000)	3 (011)	2 (010)	5 (101)	4 (100)
'10'	2 (010)	3 (011)	0 (000)	1 (001)	X (110)	X (111)
'11'	4 (100)	5 (101)	X (110)	X (111)	0 (000)	1 (001)

Tableau 5-12: Résultats d'injection de SEU sur le registre d'états "state".

state Sel	init (01)	loopState (10)	endState (11)
'01'	X (00)	endState (11)	loopState (10)
'10'	endState (11)	X (00)	init (01)

En comparant le comportement des deux descriptions, nous pouvons considérer que l'état "init" correspond à la valeur 0 du registre "counter", l'état "loopState" correspond à la suite de valeurs 1,2...4 et l'état "endState" équivaut à la valeur 5. Ceci nous permet de constater certaines dissemblances comportementales lors de la comparaison des injections effectuées dans les deux registres cibles. Par exemple, quand un SEU survient dans le registre "state" dont le contenu est l'état "loopState" alors les états erronés possibles seront "endState" et "X". Ces mêmes transitions erronées correspondent dans le registre "counter" à l'ensemble des transitions : 1 → 5, 2 → X, 3 → X, 4 → X et 4 → 5 ; les autres transitions n'ont pas d'équivalences dans le Tableau 5-12. Nous constatons aussi que les transitions "init → X" et "init → endState" n'ont pas d'équivalence dans le Tableau 5-11.

Le Tableau 5-13 résume certaines caractéristiques des graphes comportementaux générés. Pendant les simulations, nous avons observé en tout 6 signaux (externes et internes). Un maximum de 64 états peuvent donc apparaître dans le graphe généré (hormis les états "Correct" et "Latent") et 4159 transitions sont possibles après l'injection.

Tableau 5-13: Résultats obtenus après génération des graphes comportementaux.

Nombre d'États		Nombre de Transitions		Transitions communes
μ -Contrôleur	Contrôleur	μ -Contrôleur	Contrôleur	
12(19%)	13(20%)	44(1%)	52(1%)	42(81%)

Comme le montre le Tableau 5-13, le nombre d'états atteints ne dépasse pas les 20% et pas plus de 1% des transitions possibles sont observées. Par ailleurs, malgré les dissemblances comportementales mentionnées précédemment, la comparaison des résultats obtenus pour les deux descriptions montre une bonne cohérence. En particulier, le nombre de transitions communes aux graphes obtenus pour les deux niveaux de description atteint 81%, ce qui confirme les conclusions obtenues dans l'étude précédente.

En outre, il a été montré par une autre équipe que des injections de fautes sur un modèle haut niveau d'un protocole de communication en utilisant la simulation peuvent atteindre un très bon niveau de conformité avec les résultats d'injection sur une implémentation matérielle [A.A. 02]. Nous pouvons donc supposer que les conclusions de cette étude sont valables pour un grand nombre de circuits.

5.2. Injection de fautes multiples "MBF"

5.2.1. Co-processeur de multiplication entière

Des injections de fautes multiples ont été effectuées sur la description "architecture 2" du coprocesseur arithmétique étudié dans la section précédente. Deux méthodes de mutation seront appliquées sur le circuit. La première nous offre la possibilité de réaliser des injections multiples localisées à un seul registre. La deuxième nous permet de sélectionner librement les points mémoires qui seront simultanément ciblés par l'injection dans un même sous-bloc.

Le but ici n'est pas de faire une analyse détaillée des résultats du point de vue applicatif, mais d'illustrer certaines caractéristiques du processus de mutation.

5.2.1.1. Surcoûts

La première caractéristique significative concerne la complexité induite par les modifications. Ceci est particulièrement important quand les expériences sont exécutées en utilisant un prototype implémenté sur FPGA.

Avant d'évaluer l'impact des modifications sur le circuit étudié, nous précisons que ces modifications sont réalisées de telle sorte à pouvoir cibler tous les points mémoires du circuit. Dans le cas des registres de données, des inversions de bits multiples seront donc réalisées dans les registres suivant : "**counter**", "**operandB**", "**operandMQ**" et "**accumulation**". Pour le registre d'états "**state**" spécifié en utilisant un type énuméré, des transitions erronées seront

injectées. La taille de chacun des registres de donnée est égale à 16 bits et le registre "state" est défini par 5 états.

Les Tableaux 5-14 et 5-15 présentent, pour les deux processus de mutation, les surcoûts induits en fonction du degré de multiplicité. Le premier tableau illustre les résultats obtenus après synthèse par l'outil "Leonardo" de Mentor Graphics (en utilisant la bibliothèque de cellules standards "C35 0.35u" d'AMS). Le coût de la modification est évalué en terme de nombre d'E/S ajoutées, de surface, de nombre de cellules obtenues après synthèse et de chemin critique dans la netlist générée. Le deuxième tableau présente des résultats d'implémentation sur la plateforme FPGA "Virtex-II Pro" de XILINX. Ces résultats, obtenus par l'outil de synthèse "XST" de XILINX, nous permettent d'évaluer la complexité en terme de nombre d'E/S ajoutées, de nombre de regroupements de cellules logiques de base ("slices") et de chemin critique.

Tableau 5-14: Evolution de la complexité avec la multiplicité (pour une implémentation ASIC).

Multiplicité	#I/O	Surface (um2)	Surcoût (%)	# instances	Chemin critique (ns)
0(référence)	37	37419	--	240	8,90
Injection multiple limitée à un seul registre					
2	51	70179	87,5	705	9,26
3	55	82537	120,6	868	9,31
4	60	95623	155,5	1061	9,12
5	64	108927	191,1	1240	9,49
6	68	121685	225,2	1420	9,71
Injection multiple non limitée à un seul registre					
2	54	80917	116,2	884	10,92
3	61	98953	164,4	1140	16,03
4	69	119720	219,9	1468	19,14
5	76	134971	260,7	1668	24,42
6	83	165893	343,3	1904	26,59

Tableau 5-15: Evolution de la complexité avec la multiplicité (en ressources FPGA).

Multiplicité	#I/O	# slices	Surcoût (%)	Chemin critique (ns)
0(référence)	37	103	--	4,51
Injection multiple limitée à un seul registre				
2	51	343	233	5,64
3	55	463	349,5	6,21
4	60	543	427,2	6,27
5	64	562	445,6	6,92
6	68	705	584,5	7,49
Injection multiple non limitée à un seul registre				
2	54	351	240,8	11,26
3	61	434	321,3	16,18
4	69	509	394,2	21,25
5	76	575	458,3	26,52
6	83	638	519,4	33,21

Dans le cas d'une implémentation ASIC, les courbes des Figures 5-4 et 5-5 illustrent respectivement l'évolution de la surface et du chemin critique en fonction de la multiplicité. Nous pouvons constater que la complexité augmente rapidement avec le degré maximum de multiplicité, particulièrement lorsque les fautes multiples ne se limitent pas à un seul registre. Dans ce cas, le chemin critique augmente aussi particulièrement rapidement (ce qui limite la fréquence d'horloge d'un prototype, et augmente donc le temps nécessaire pour exécuter les expériences). Lorsque les fautes sont limitées à un seul registre, les surcoûts sont plus faibles, voire négligeables du point de vue de la fréquence de fonctionnement.

Dans le cas d'une implémentation sur le support FPGA, les courbes des Figures 5-6 et 5-7 illustrent respectivement l'évolution du nombre des cellules de base et du chemin critique en fonction de la multiplicité. Nous remarquons que le surcoût en ressources FPGA augmente rapidement pour les deux méthodes de mutation. Cependant, pour une approche d'injection limitée à un seul registre, les surcoûts en terme de fréquence de fonctionnement sont beaucoup plus faibles que pour une injection plus générale.

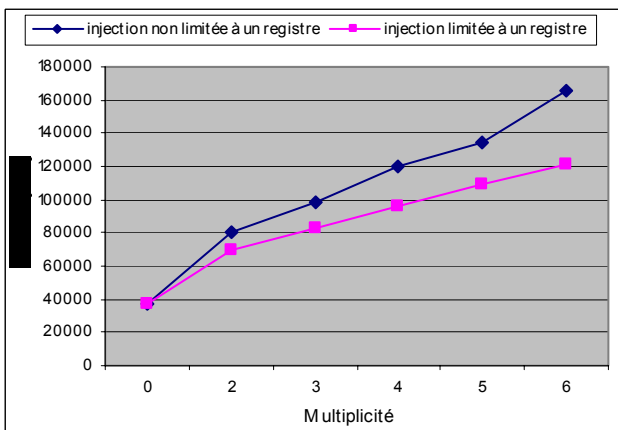


Figure 5-4: Evolution de la surface en fonction de la multiplicité (implémentation ASIC)

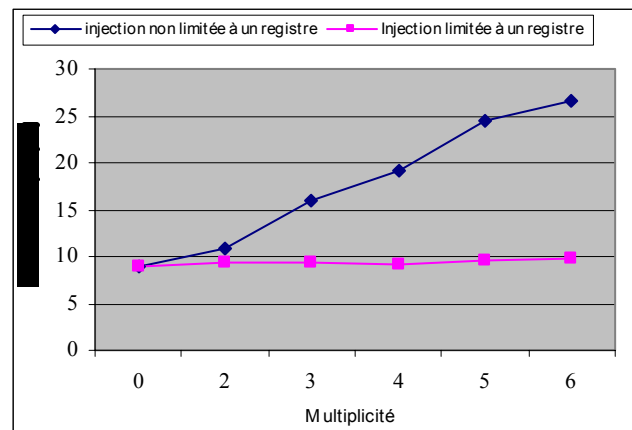


Figure 5-5: Evolution du chemin critique en fonction de la multiplicité (implémentation ASIC)

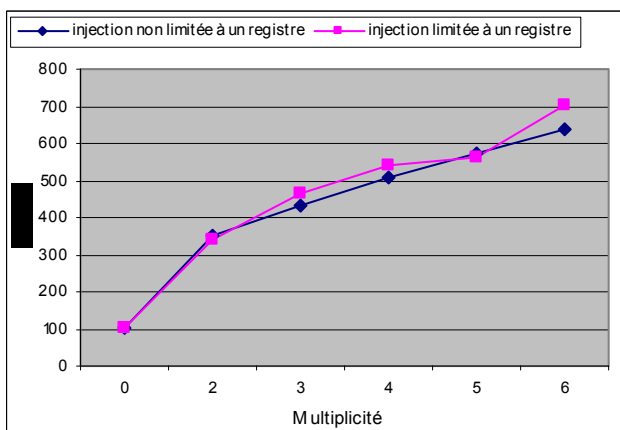


Figure 5-6: Evolution du nombre de cellules "slices" en fonction de la multiplicité (implémentation FPGA)

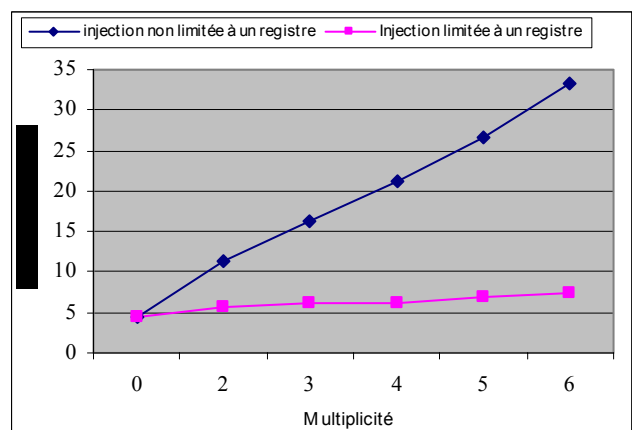


Figure 5-7: Evolution du chemin critique en fonction de la multiplicité (implémentation FPGA)

5.2.1.2. Expériences d'injection

Les expériences d'injection, basées sur des simulations RTL avec ModelSim de Mentor Graphics, exécutent un programme de test fonctionnel de 183 cycles. Une seule injection multiple est réalisée par expérience et n'importe quel élément mémoire du circuit peut être ciblé. Les cycles d'injection ont été choisis de manière exhaustive (de 5 à 182).

Concernant la distribution spatiale, les expériences effectuées incluent des injections de transitions erronées dans le registre d'états et des injections aléatoires de MBFs dans les registres de données. La stratégie dépend de la méthode d'injection appliquée. Dans le cas de la méthode restrictive, 5 transitions erronées sont injectées dans le registre d'états et 10 injections aléatoires de MBF sont effectuées dans chacun des 4 registres de données pour chaque cycle d'horloge (cela nous fait un total de 8010 expériences). Dans le cas de la méthode non restrictive, nous effectuons un total de 45 injections aléatoires pour chaque cycle de simulation. Une injection donnée peut inclure l'injection d'une transition erronée dans le registre d'états et/ou un MBF distribué aléatoirement dans les 4 registres de données (ce qui nous fait aussi un total de 8010 expériences). Pour chacune des deux méthodes d'injection, nous réalisons 5 campagnes d'injection correspondant au degré de multiplicité allant de 2 jusqu'à 6.

La dernière campagne d'injection a consisté à injecter des inversions de bit uniques (multiplicité spatiale égale à 1) et des transitions erronées dans les mêmes cibles que précédemment. Elle a été effectuée de manière exhaustive (cycle et cible d'injection), ce qui nous fait un total de $(182-5+1)*(5+16*4) = 12282$ expériences.

- **Analyse des résultats**

L'objectif est ici d'évaluer l'impact des injections de fautes multiples sur les résultats de l'analyse. Les graphes comportementaux générés montrent les propagations d'erreur enregistrées sur les signaux externes (le bus de données "busout" et le signal "endofmult") et deux registres internes (le registre d'états "state" et le compteur de boucle "counter"). Afin d'évaluer l'impact des fautes de type MBFs par rapport aux SEUs, nous comparons chacun des graphes issus de l'analyse des expériences d'injection de fautes de multiplicité supérieure à 1 avec celui issu des expériences d'injection de fautes simples. Les caractéristiques des graphes générés en fonction des différentes valeurs de la multiplicité sont illustrées dans le Tableau 5-16. Ces résultats indiquent le nombre de configurations erronées différentes observées pendant les expériences (états), et le nombre de transitions entre ces états. Nous illustrons aussi les états et les transitions communs aux deux graphes (colonne 3) ainsi que les états et les transitions spécifiques à chacun des graphes (colonnes 4 et 5).

Tableau 5-16: Résultat d'analyse de la propagation d'erreur pour les différentes valeurs de multiplicité.

Multiplicité	Etats/Transitions	Etats/Transitions communs	Etats/Transitions spécifiques (MBF)	Etats/Transitions spécifiques (SEU)
1	20/135	--	--	--
Injection multiple limitée à un seul registre				
2	20/137	20/135	0/2	0/0
3	20/136	20/134	0/2	0/1
4	20/136	20/134	0/2	0/1
5	20/137	20/134	0/3	0/1
6	22/141	20/134	2/7	0/1
Injection multiple non limitée à un seul registre				
2	22/180	20/132	2/48	0/3
3	22/187	20/129	2/58	0/6
4	22/184	20/124	2/60	0/11
5	22/183	20/124	2/59	0/11
6	22/180	20/121	2/59	0/14

La complexité globale des propagations, illustrée par le nombre de transitions dans un graphe, ne dépend pas beaucoup pour cet exemple simple de la multiplicité mais dépend plutôt de la limitation spatiale. En outre, les résultats d'injection des inversions de bits uniques ou multiples limitées à un seul registre sont très semblables. La différence est déjà plus nette pour des fautes multiples réparties dans différents registres. Lorsque les fautes multiples ne peuvent pas être supposées limitées à un registre, il est donc utile de réaliser des injections de MBF, au moins pour une multiplicité faible.

5.2.2. Co-processeur de multiplication modulaire

5.2.2.1. Description

Des campagnes d'injections de fautes ont également été effectuées sur un co-processeur de calcul utilisé pour accélérer un chiffrement RSA (multiplieur de Montgomery). La multiplication de Montgomery (ou multiplication modulaire) consiste à calculer le produit $A*B*R^{-1} \text{ Mod } N$ avec $R=2^n$, A et B étant deux nombres codés sur n bits et N étant le modulo codé sur n bits.

L'architecture générale du multiplieur de Montgomery est illustrée en Figure 5-8. Chacun des blocs a un rôle spécifique :

- **Interface** : se charge de gérer la communication entre le multiplieur et le monde extérieur. C'est à travers elle que va se faire l'échange de données, et plus particulièrement l'écriture de chacun des opérandes (A, B et N) et la lecture du résultat.
- **Décalage** : synchronise l'envoi des données (envoi de A bit par bit) vers le bloc de calcul.
- **Registres de données** : utilisés pour stocker les opérandes et le résultat de calcul.
- **Contrôle** : se charge de lancer et de terminer le calcul et, suivant l'implémentation, de déterminer si une soustraction est nécessaire en fin de calcul (calcul du résultat modulo).

- **Calcul** : c'est le cœur du multiplieur, constitué d'un bloc combinatoire où sont réalisés les calculs.

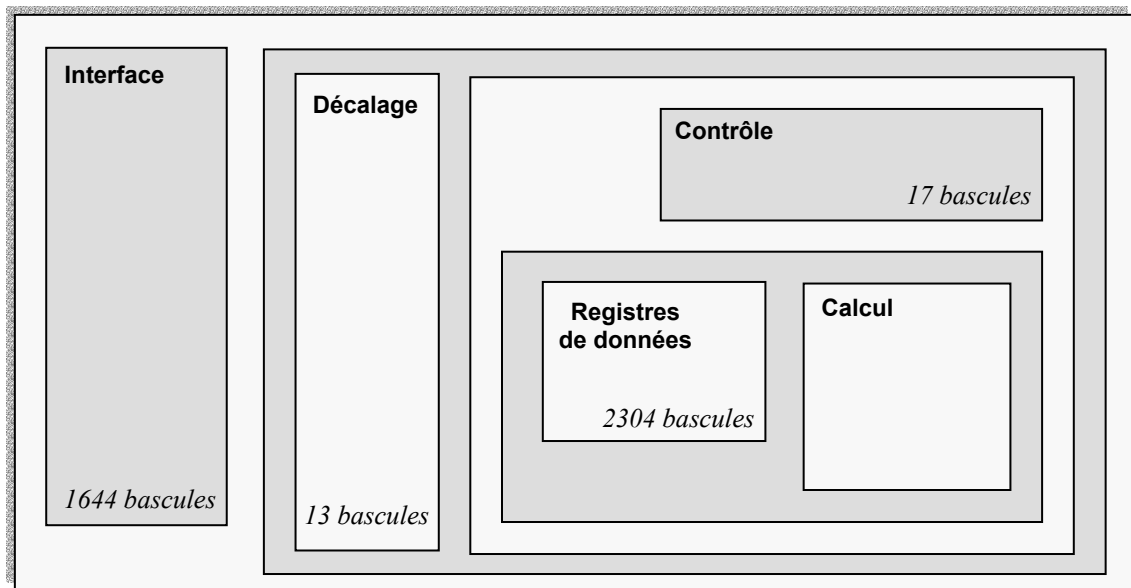


Figure 5-8: Schéma bloc du multiplieur de Montgomery

5.2.2.2. Expériences d'injection

Les expériences d'injection ont été réalisées avec les mêmes moyens que pour l'exemple précédent. Des mutants hétérogènes ont été générés pour la description globale du circuit ; ils permettent de réaliser des injections multiples sans limitation spatiale, avec des modèles de fautes pouvant être différents d'un bloc à l'autre dans la hiérarchie. Tous les éléments mémoires du circuit peuvent être ciblés par l'injection ; ils sont regroupés par bloc dans le Tableau 5-17.

Tableau 5-17: Définition par bloc des cibles d'injection

Bloc	# registres	Type	Taille	Modèle de fautes	Total
Interface	1	Enuméré	3 états	Trans. err.	3
	6	Binaire	1 bit	MBF	1644
	3	Binaire	2 bits		
	2	Binaire	32 bits		
	1	Matrice	49*32 bits		
Décalage	3	Binaire	1 bit	MBF	13
	1	Entier	10 bits		
Contrôle	6	Binaire	1 bit	MBF	17
	1	Entier	11 bits		
Registres de données	1	Binaire	2304 bits	MBF	2304

Les fautes injectées pendant les expériences modélisent les phénomènes SEU et MBF au niveau RTL. Dans le cas de registres de type binaire ou de type entier (définis dans le bloc "Décalage" et "Contrôle"), des inversions de bits sont réalisées dans les mémoires et les bascules. Dans le cas d'un registre d'états (défini dans le bloc "Interface") spécifié en utilisant un type énuméré, des transitions erronées sont injectées.

Les campagnes d'injection ont été définies de telle sorte à atteindre un bon compromis entre la précision des résultats et la durée des expériences. Le nombre de fautes injectées à chaque campagne est limité à 2000. Cette limitation est due également à la durée des expériences car chaque expérience prend en moyenne 19 secondes sur une station "SunBlade" 150 (ceci correspond à environ 11 heures par campagne d'injection). Les injections, effectuées pendant l'exécution d'un programme de 2852 cycles, surviennent après la remise à zéro des registres (effectuée au 5ème cycle). Pour une expérience d'injection donnée, une seule injection est effectuée ; le programme d'exécution de la campagne choisit aléatoirement un numéro de cycle d'injection, un numéro de bloc ainsi que les indices des bits à inverser dans le registre virtuel correspondant.

La qualité de l'analyse comportementale du circuit dépend des points d'observation choisis par le concepteur. Les points les plus significatifs observés pendant les expérimentations correspondent à deux bus et sept signaux ; ceci conduit à un maximum de 513 états possibles et 263170 transitions possibles dans le graphe de propagation d'erreurs.

- **Analyse des résultats**

La simulation comportementale a l'avantage d'être beaucoup plus rapide que la simulation au niveau portes. Cependant, la description comportementale d'un circuit reste pauvre en détails, par rapport à l'implémentation finale. Dans la section 5.1, nous avons démontré que des informations très significatives peuvent toutefois être obtenues à partir d'une description de haut niveau du circuit.

Par ailleurs, parmi les différentes expériences d'injection de fautes basées sur la simulation au niveau RTL du circuit étudié, certaines expériences se sont interrompues au milieu de l'exécution. Ceci est dû au fait que des fautes simples et multiples ont amené des variables entières (incluses dans les blocs "Décalage" et "Contrôle") à prendre une valeur hors de leur intervalle, ce qui a provoqué l'arrêt de la simulation, et a rendu donc impossible l'analyse de la propagation d'erreur après cet événement, même si le circuit réel ne s'arrêterait pas. Par conséquent des états de type "crash" sont ajoutés automatiquement dans les graphes générés. Le Tableau 5-18 illustre les pourcentages de cas conduisant à l'état "crash" obtenus lors des différentes campagnes d'injection (2000 injections par campagne). Il est visible que ce pourcentage tend à augmenter avec la multiplicité des fautes.

Tableau 5-18: Pourcentages des cas conduisant à l'état "crash" pour différentes valeurs de la multiplicité

Multiplicité	1	2	3	4	5	6
% "Crash"	2,5	6,6	9,3	12,3	15,4	18

Concernant les modèles comportementaux générés, le Tableau 5-19 résume des résultats obtenus en injectant des inversions multiples (avec une multiplicité allant de 2 à 6). Ces résultats indiquent le nombre d'états erronés différents observés pendant les expériences, et le nombre de transitions entre ces états (représentatives des propagations d'erreur dans le circuit). La part des états et des transitions commune ou spécifique à une analyse avec fautes multiples est représentée, par référence aux états et transitions observés pour des injections avec multiplicité égale à 1. Ces résultats montrent clairement l'influence de la multiplicité des erreurs injectées sur les conséquences au niveau de l'application. Les différences sont beaucoup plus sensibles que pour le premier exemple étudié, ce qui peut s'expliquer par la plus grande complexité de ce deuxième circuit.

Tableau 5-19: Résultats de l'analyse de propagation d'erreurs pour différentes valeurs de la multiplicité

Multiplicité	Etats/Transitions	Etats/Transitions communs	Etats/Transitions spécifiques (MBF)	Etats/Transitions spécifiques (SEU)
1	31/69	--	--	--
Injection multiple sans limitation spatiale				
2	39/95	30/67	9/28	1/2
3	37/89	30/65	7/24	1/4
4	39/94	30/63	9/31	1/6
5	48/121	31/65	17/56	0/4
6	48/122	31/65	17/57	0/4

Etats/Transitions communs : états et transitions obtenus à la fois avec des injections simples et multiples

Etats/Transitions spécifiques : états et transitions spécifiques au cas des inversions de bit multiples

Etats/Transitions spécifiques (SEU) : états et transitions spécifiques au cas des inversions de bit uniques

5.3. Conclusion sur les expériences d'injection

Les résultats obtenus montrent l'intérêt d'une validation des caractéristiques de sûreté dès les plus hauts niveaux de description. Les résultats qualitatifs obtenus en termes de chemins de propagation d'erreur montrent une très bonne corrélation entre des analyses exécutées à différents niveaux de spécification du circuit et en utilisant différents niveaux de modélisation d'erreurs ou de fautes. De tels résultats peuvent aider le concepteur à déterminer très tôt quel type de comportement est à considérer quand une faute donnée se produit. Par ailleurs, une telle analyse peut être très utile pour identifier très tôt les points optimaux où des mécanismes de protection devraient être insérés pour le durcissement par conception.

Les résultats expérimentaux sur l'injection de fautes de type MBF montrent que si, pour une application donnée, des fautes multiples peuvent réellement se produire, il est important d'en tenir compte (au moins avec une petite multiplicité). En effet, faire l'hypothèse de

l'inversion d'un bit unique conduit à négliger potentiellement un nombre notable de propagations d'erreurs possibles dans un circuit. Ceci peut conduire à surestimer gravement la robustesse d'un circuit face à différents types de fautes. L'approche proposée permet d'étendre les analyses habituelles afin de répondre à ce problème. La limitation principale reste l'augmentation notable du temps nécessaire pour réaliser les expérimentations, à cause de l'explosion combinatoire du nombre de combinaisons de fautes. Il est donc nécessaire pour le concepteur de réduire ce nombre de combinaisons en prenant en compte par exemple certaines caractéristiques géométriques (plan de masse) ou structurelles (connexions entre blocs).

Conclusion et perspectives

La conception de circuits VLSI sûrs exige des techniques et des outils pour une évaluation prédictive du niveau de robustesse atteint contre les effets des fautes envisagées dans l'application. En particulier, les concepteurs sont intéressés par des outils capables de réaliser des campagnes d'injection de fautes avant que le circuit ne soit fabriqué. Les techniques d'injection de fautes basées sur la modification d'une description de haut niveau offrent des perspectives prometteuses. Elles permettent, lorsque l'on dispose uniquement du modèle d'un système, d'obtenir une évaluation "très tôt" de la sûreté de fonctionnement de ce système. Par ailleurs, elles sont très flexibles en terme de modèle de faute puisque tous les modèles de fautes peuvent potentiellement être supportés et les fautes peuvent être injectées dans n'importe quel module du système. Leur principale limitation, à savoir le temps de simulation requis, peut être réduite en utilisant une approche à base de prototypage.

Le flot d'analyse actuellement automatisé dans notre environnement part d'une spécification de haut niveau d'abstraction (VHDL RTL), et grâce à des outils automatiques, permet à l'utilisateur de définir les différents paramètres de la campagne d'injection, de modifier la spécification VHDL, de contrôler l'exécution de la campagne par simulation du circuit et d'analyser les résultats obtenus. En effet, le flot offre la possibilité de générer un modèle comportemental détaillé du circuit, montrant les chemins de propagation d'erreurs avant occurrence d'une défaillance ou d'une détection.

La génération d'un modèle détaillé de propagation d'erreurs est la première contribution originale de l'environnement proposé. Cette analyse peut être vue comme une extension de la classification, recherchant non seulement l'état final du circuit (défaillance, détection, ...), mais aussi les états internes activés pendant les expériences et qui ont mené à cet état final. Les informations fournies par cette analyse permettent d'identifier les points optimaux où des mécanismes de protection devraient être insérés pour le durcissement.

Nous nous sommes intéressé également à la modélisation à haut niveau des effets de fautes transitoires un circuit. La deuxième contribution de ce travail est l'utilisation de mutations optimisées pour l'instrumentation d'une description comportementale de circuit. Les mutants générés permettent non seulement de réaliser des simulations au niveau RTL, d'où un gain notable de temps par rapport aux simulations niveau portes, mais ils prennent encore en compte les contraintes spécifiques à l'émulation. Le processus de génération de ces mutants a été détaillé pour les différents modèles de fautes visés. Nous avons commencé par définir une approche de génération de mutants permettant d'effectuer des injections de fautes simples dans les points mémoire du circuit, tout en étant synthétisables de façon efficace. Les mutants générés prenaient en compte le modèle des inversions de bit uniques et le modèle des transitions

erronées afin de représenter à différents niveaux d'abstraction le phénomène SEU. Nous avons également démontré la faisabilité de l'automatisation de la génération contrôlée de mutants VHDL pour l'injection des transitions erronées. Les résultats ont prouvé que le type d'instrumentation a un impact important sur une campagne d'injection basée sur l'émulation, car les différentes modifications peuvent mener à des caractéristiques de prototypage du circuit très différentes. Ensuite, nous avons étendu ce modèle classique d'inversions de bits uniques aux inversions multiples. La génération de mutants VHDL pour ce modèle étendu est présentée en détail en montrant comment plusieurs modèles de fautes peuvent être combinés et inclus dans une seule campagne d'injection, par le biais d'une génération hétérogène de mutants. Dans le cas d'expériences basées sur l'émulation, des compromis entre la complexité et la généralité sont également étudiés.

Des résultats expérimentaux, obtenus à l'issue de campagnes d'injection de fautes simples sur un coprocesseur arithmétique et un processeur Fuzzy, montrent l'intérêt d'une analyse des caractéristiques de sûreté dès les plus hauts niveaux de description. Des résultats pertinents montrent une très bonne corrélation entre des analyses exécutées à différents niveaux de spécification du circuit et en utilisant différents niveaux de modélisation d'erreurs ou de fautes. De tels résultats peuvent aider le concepteur à déterminer très tôt quel type de comportement est à considérer quand une faute donnée se produit.

Des expériences d'injection de fautes multiples ont été effectuées sur le coprocesseur arithmétique et un coprocesseur de multiplication modulaire. Les résultats montrent que si, pour une application donnée, des fautes multiples peuvent réellement se produire, il est important d'en tenir compte (au moins avec une petite multiplicité). En effet, faire l'hypothèse de l'inversion d'un bit unique conduit à négliger potentiellement un nombre notable de propagations d'erreurs possibles dans un circuit. Ceci peut conduire à surestimer gravement la robustesse d'un circuit face à différents types de fautes.

➤ **Perspectives**

Une des limites de la méthode de génération de mutants pour les inversions multiples est liée à l'injection spatiale dans le cas d'une description hiérarchique. En effet, les bits concernés par les inversions simultanées doivent actuellement être confinés dans un sous-bloc donné. Dans certains cas (plus généraux) où les fautes peuvent se répartir librement dans le circuit, il serait nécessaire d'étendre l'approche pour permettre d'injecter des fautes multiples simultanément sur l'ensemble du circuit et ainsi améliorer la qualité globale de l'analyse de sûreté. Un des principaux objectifs sera d'optimiser le nombre d'entrées primaires additionnelles.

L'autre limitation reste l'augmentation notable du temps nécessaire pour réaliser les expérimentations, à cause de l'explosion combinatoire du nombre de combinaisons de fautes. Il est donc nécessaire pour le concepteur de réduire ce nombre de combinaisons en prenant en compte par exemple certaines caractéristiques géométriques (plan de masse) ou structurelles

(connexions entre blocs). Une autre approche proposée dans [R.L. 05] se base sur la combinaison de la vérification formelle et de la génération contrôlée de mutants.

Un autre objectif serait d'étendre le flot d'analyse afin d'obtenir un lien automatisé avec des outils de durcissement. Ceci favorisera une validation plus rapide des propriétés de sûreté du circuit.

Par ailleurs, pour faire face à la complexité des circuits, il est important de généraliser l'étude des propagations d'erreur à la génération de modèles hiérarchiques, de telle sorte que, par exemple, le modèle obtenu pour un bloc d'IP puisse être directement utilisé comme un sous-modèle pour le circuit qui intègre cet IP.

Enfin, il serait intéressant de prévoir à terme des expériences pour comparer les résultats des prédictions avec des résultats sur le circuit soumis à un flux de particules ou à des attaques par laser.

Annexe A

- Description "micro-contrôleur" du processeur Fuzzy

```
entity Control1_nc is
  generic(in_number: integer := 2; j_pipe: integer := 6);

  port( clk : in std_logic;
        reset : in std_logic;
        dir2 : out std_logic_vector(in_number downto 1);
        me1 : out std_logic;
        me2 : out std_logic;
        pipeline : out std_logic;
        clear_memory : out std_logic;
        accumulation : out std_logic;
        -- Ports d'entrée/sortie nécessaires pour l'injection
        en_seu : in std_logic;          -- autorise l'injection
        asyn_inj : in std_logic;        -- inverser le bit sélectionné
        Sel : in std_logic_vector(1 downto 0); -- Sélectionner le bit à inverser
        counter_out : out integer range 0 to j_pipe - 1
  );
end Control1_nc;

architecture FPGA of Control1_nc is
  signal counter: integer range 0 to j_pipe - 1;
  signal counter2: std_logic_vector(in_number downto 1);
  signal int_pipe: std_logic;
  signal clk_reg: std_logic ; -- register clock
begin

  P_counter:process(reset,clk_reg)
  variable converted: std_logic_vector(2 downto 0);
  begin
    if (reset = '1') then
      counter <= j_pipe - 1;
    elsif (clk_reg'event and clk_reg = '0') then
      if(en_seu='0') then
        if (counter = j_pipe - 1) then
          counter <= 0;
        else
          counter <= counter + 1;
        end if;
      else
        converted := CONV_STD_LOGIC_vector(counter,3);
        case Sel is
          when "01"=>converted(0) := not converted(0);
          when "10"=>converted(1) := not converted(1);
          when "11"=>converted(2) := not converted(2);
          when others => null;
        end case;
        counter <= conv_integer(converted);
      end if;
    end if;
  end process P_counter;

  P_pipeline:process
  begin
    wait on clk until clk = '1';
    if (counter = 0) then
```

```
        int_pipe <= '1';
    else
        int_pipe <= '0';
    end if;
end process P_pipeline;

P_counter2:process
begin
    wait on clk until clk = '0';
    if (int_pipe = '1') then
        counter2 <= (others => '0');
    else
        counter2 <= counter2 + "01";
    end if;
end process P_counter2;

P_accumulation:process
begin
    wait on clk until clk = '1';
    if (counter > 0 and counter <= 2**in_number) then
        accumulation <= '1';
    else
        accumulation <= '0';
    end if;
end process P_accumulation;

dir2 <= counter2;
me1    <= int_pipe;
me2    <= clk;
pipeline <= int_pipe;
clear_memory <= int_pipe;
----- injection asynchrone-----
seq2: process(Clk,asyn_inj)
begin
    if(asyn_inj='1') then
        clk_reg <= not(Clk) ;
    else
        clk_reg <= Clk;
    end if;
end process ;
----- Fin -----
counter_out<=counter;

end FPGA;
```

Annexe B

- Description "contrôleur" du processeur Fuzzy

entity Control1_nc is

generic(in_number: **integer** := 2; j_pipe: **integer** := 6);

port(clk: in **std_logic**; -- Senhal de reloj
reset: in **std_logic**; -- A 1 activo, a 0 desactivo
dir2: **out std_logic_vector**(in_number **downto** 1);
me1: **out std_logic**;
me2: **out std_logic**;
pipeline: **out std_logic**;
clear_memory: **out std_logic**;
accumulation: **out std_logic**;
-- Ports d'entrée/sortie nécessaires pour l'injection
en_seu :**in std_logic**; -- autoriser l'injection
asyn_inj : **in std_logic**; -- inverser le bit sélectionné
Sel :**in std_logic_vector**(1 **downto** 0); -- Sélectionner le bit a inverser
state_out: **out std_logic_vector** (1 **downto** 0)

);

end Control1_nc;

architecture FPGA of Control1_nc is

signal state, nstate: **std_logic_vector**(1 **downto** 0) ; -- current and next state

constant init: **std_logic_vector**(1 **downto** 0):="01";
constant loopState: **std_logic_vector**(1 **downto** 0):="10";
constant endState: **std_logic_vector**(1 **downto** 0):="11";
signal int_loop, nint_loop: **integer range** 0 to j_pipe - 1;
signal counter2, ncounter2: **std_logic_vector**(in_number **downto** 1);
signal int_pipe, nint_pipe: **std_logic**;
signal naccumulation: **std_logic**;
signal clk_reg: **std_logic** ; -- register clock

begin

P_counter:**process**(reset,clk_reg)

begin

if (reset = '1') **then**
state <= endState;
elsif (clk_reg'event and clk_reg = '0') **then**
int_loop <= nint_loop;
counter2 <= ncounter2;
if(en_seu='0') **then**
state <= nstate ;
else
case Sel **is**
when "01" => state(0) <= **not** state(0);
when "10" => state(1) <= **not** state(1);
when others => **null**;

end case;

end if ;

end if;

end process P_counter;

P_acc_int:**process**

begin

wait on clk **until** clk = '1';

```
    int_pipe <= nint_pipe;
    accumulation <= naccumulation;
end process P_acc_int;

comb: process (state,int_loop,counter2)
begin
    case state is
        when init =>
            nstate <= loopState;
            naccumulation <= '0';
            nint_loop <= 0;
            nint_pipe <= '1';
            ncounter2 <= (others => '0');

        when loopState =>
            if (int_loop <= 2**in_number-2) then
                nstate <= loopState;
            else
                nstate <= endState;
            end if;
            naccumulation <= '1';
            nint_loop <= int_loop + 1;
            nint_pipe <= '0';
            ncounter2 <= counter2 + "01";

        when endState =>
            nstate <= init;
            naccumulation <= '0';
            nint_loop <= 0;
            nint_pipe <= '0';
            ncounter2 <= counter2 + "01";

        when others => null;
    end case ;
end process ;

dir2 <= counter2;
me1 <= int_pipe;
me2 <= clk;
pipeline <= int_pipe;
clear_memory <= int_pipe;
state_out <= state;
----- injection asynchrone-----
seq2: process(Clk,asyn_inj)
begin
    if(asyn_inj='1') then
        clk_reg <= not(Clk) ;
    else
        clk_reg <= Clk;
    end if;
end process ;
----- Fin -----
end FPGA;
```

Références bibliographiques

- [A.A. 97] A. Amendola, L. Impagliazzo, P. Marmo, F. Poli, "Experimental Evaluation of Computer-Based Railway Control Systems", in Proc. 27th Int. Symp. on Fault-Tolerant Computing (FTCS-27), Seattle, WA, USA, June 1997, pp. 380-384.
- [A.A. 02] A. Ademaj, P. Grillinger, P. Herout, and J. Hlavicka, "Fault Tolerance Evaluation Using Two Software Based Fault Injection Methods", in 8th IEEE International On-Line Testing Workshop, Isle of Bendor, France, July 8-10, 2002, pp. 21-25.
- [A.A. 04-a] A. Avizienis, J. C. Laprie, B. Randell, "Dependability and its Threats: A Taxonomy", in Proceedings of the Building the Information Society: Proc. IFIP 18th World Computer Congress, Toulouse, France Jacquart, 22-27 August, 2004, R. (ed) pp. 91-120.
- [A.B. 99] A. Benso, P. Prinetto, M. Rebaudengo, M. Reorda, "EXFI: A Low-cost Fault Injection System for Embedded Microprocessor-Based Boards", ACM Trans. On Design Automation of Electronic Systems, Vol. 3, No. 4, October 1998, Pages 626-634.
- [A.C. 99] A. Cataldo, "Intel Scans for Soft Errors in Processor Designs", EETimes, 06/16/99
- [A.D. 81] A. Dantec, "Le phénomène de latchup dans les circuits integres CMOS", Toute l'électronique, TLE, 1981, 465, p.45.
- [A.F. 00] A. Fin, F. Fummi, "A VHDL error simulator for functional test generation", Design, Automation and Test in Europe Conference (DATE), March 27-30, 2000, pp. 390-395.
- [A.M. 96] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "Fault behaviour observation of a microprocessor system through a VHDL simulation-based fault injection experiment", in Design Automation Conference, September 1996, pages 536-541.
- [B.P. 00] B. Parrotta, M. Rebaudengo, M. S. Reorda, and M. Violante. "New techniques for accelerating fault injection in VHDL description", in International On-Line Testing Workshop, July 2000, pages 61-66.
- [C.C. 02] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits", Proc. Int'l Conf. Dependable Systems and Networks (DSN-2002), 2002, pp. 205-209.
- [C.F. 04] Centauri-FTL Systems, Unified Compiler/Simulator Suite VHDL, VHDL-AMS, VHDL-X, VHDL-RF/MW, Verilog, Verilog-AMS, System Verilog, SPICE & C++ (via PLI). Disponible sur le site web suivant : <http://www.ftlsystems.com/products/centauri.shtml>.
- [C.P. 01] C. Poivey, J.W. Howard, S. Buchner, K.A. LaBel, J.D. Forney, H.S. Kim, A. Assad, "Development of a Test Methodology for Single Event Transients (SETs) in Linear Devices", IEEE Trans. On Nucl. Sci., vol. 48, n°6, Dec. 2001, pp. 2180-2186.

- [C.P. 03] C. Poivey, S. Buchner, J.W. Howard, K.A. LaBel, "Testing Guidelines for Single Event Transient (SET): Testing of Linear Devices", Technical report, NASA Goddard Space Flight Center, http://radhome.gsfc.nasa.gov/radhome/papers/2003_Linear.pdf, June 30, 2003.
- [C.W. 90] C. J. Walter, "Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault Tolerance", IEEE Trans. Reliability, Oct. 1990, vol. 39, no. 4, pp. 492-499.
- [D.A. 02] D. Alexandrescu, L. Anghel, M. Nicolaidis, "New methods for evaluating the impact of single event transients in VDSM Ics", Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Vancouver, Canada, November 6-8, 2002, pp. 99-107.
- [D.G. 00] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "A study of the effects of transient fault injection into the VHDL model of a fault-tolerant microcomputer system", in IEEE International On-Line Testing Workshop, July 03-05, 2000, pages 73-79.
- [D.P. 94] D. Powell, "Distributed Fault-Tolerance-Lessons from Delta-4", IEEE Micro, vol. 14, no. 1, Feb. 1994, pp. 36-47.
- [D.S. 92] Daniel P. Siewiorek, Robert S. Swarz, "Reliable Computer Systems, Design and Evaluation", second edition, 1992.
- [E.F. 96-a] E. Fuchs, "An Evaluation of the Error Detection Mechanisms in MARS using Software Implemented Fault Injection", in Proc. 2nd European Dependable Computing Conference (EDCC-2), Taormina, Italy, October 1996, pp. 73-90.
- [E.F. 96-b] E. Fuchs, "Software Implemented Fault Injection", PhD dissertation, Vienna Univ. of Technology, Austria, 1996.
- [E.J. 94-a] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: A fault injection tool and some preliminary experimental results", In IEEE International Workshop on Integrating Error Models with Fault Injection, April 1994, pp. 13-14.
- [E.J. 94-b] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool", in 24th Symposium on FTCS, June 1994, pp. 66-75.
- [E.N. 96] E. Normand, "Single Event Upset at Ground Level", IEEE Trans. Nuclear Science, vol. 43, no. 6, Feb. 1996, pp. 2742-2750.
- [E.P. 83] E.L. Petersen, "Single Event Upsets in Space: Basic Concepts", Tutorial Short Course, IEEE NSREC'83, 1983.
- [F.D. 02] F. DARRACQ, T. BEAUCHENE, V. POUGET, H. LAPUYADE, D. LEWIS, P. FOUILLAT, A. TOUBOUL, "Single-Event sensitivity of a single SRAM cell", IEEE Transactions Nuclear Science, vol 49(3), 2002, pp1496-1490.
- [F.S. 96] F. W. Sexton, "Microbeam studies of single-event effects", IEEE Trans. Nucl. Sci., vol. 43, Apr. 1996, pp. 687-695.

- [F.V. 00] F. Vargas, A. Amory, and R. Velazoo, "Estimation circuit fault-tolerance by means of transient-fault injection in vhdl", in IEEE International On-Line Testing Workshop, July 2000, pages 67-72.
- [G.A. 96] G. Al Hayek, C. Robach, "From specification validation to hardware testing: a unified method", International Test Conference (ITC), 1996, pp. 885-893.
- [G.A. 99] G. Al Hayek, C. Robach, "From design validation to hardware testing: a unified approach", Journal of Electronic Testing: Theory and Applications (JETTA), vol. 14, 1999, pp. 133-140.
- [G.C. 92] Gwan S. Choi and Ravishankar K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis", IEEE Transactions on Computers, vol. 41, No. 12, December 1992, pp. 1515-1526.
- [G.K. 92] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", in Proc. of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), July 1992, pp. 336-344.
- [G.K. 95] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", IEEE Trans. on Computers, vol. 44, no. 2, Feb. 1995, pp. 248-260.
- [G.S. 01] G. Snider, B. Shacklefort, and R. J. Carter, "Attacking the semantic gap between application programming languages and configurable hardware", in ACM International Symposium on Field Programmable Gate Arrays, 2001, pages 115-124.
- [H.H. 65] H. D. Habing, "The use of lasers to simulate radiation induced transients in semiconductor devices and circuits", IEEE Trans. Nucl. Sci., vol. NS-12, Dec. 1965, pp. 91-100.
- [H.M. 94] H. Madeira, M. Rela, F. Moreira and J. G. Silva, "RIFLE: A General Purpose Pin-level Fault Injector", in Proc. 1st European Dependable Computing Conf. (EDCC-1), (Berlin, Germany), Springer-Verlag, 1994, pp.199-216.
- [J.A. 90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation-A Methodology and Some Applications", IEEE Trans. Software Eng., vol. 16, no. 2, Feb. 1990, pp. 166-182.
- [J.A. 91] J. Arlat, Y. Crouzet and J.-C. Laprie, "Fault Injection for the Experimental Validation of Fault Tolerance", in Proc. Esprit Conference'91, (CEC-DGXIII), pp.791-805, Brussels, Belgium, November 1991, pp.791-805.
- [J.A. 99] J. Arlat, J. Boué, Y. Crouzet, "Early validation of fault-tolerant systems: fault injection into VHDL models", LAAS research report n° 98.392, September 1998, revised January 1999.
- [J.B. 00] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A prototype of a vhdl-based fault injection tool", in IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, October 2000, pages 396-404.

- [J.C. 95] J. A. Clark, D. K. Pradhan, "Fault Injection A Method For Validating Computer-System Dependability", in *Computer*, Vol. 28, No. 6, June 1995, pp. 47-56.
- [J.C. 98] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Trans. Software Eng.*, vol. 24, no. 2, Feb. 1998, pp. 125-136.
- [J.G. 95] J. Guthoff, V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method", *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995, pp. 196-206.
- [J.G. 01] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil, "Comparison and application of different vhdl-based fault injection techniques". In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, San Francisco, California, USA, October 2001, pages 233-241.
- [J.H. 96] J. H. Hong, S. A. Hwang, and C. W. Wu, "An FPGA-based hardware emulator for fast fault emulation", In *IEEE Midwest Symposium on Circuits and Systems*, vol. 1, August 1996, pp. 345-348.
- [J.K. 91] J. Karlsson et al., "Two fault injection techniques for test of fault handling mechanisms", *ITC*, 1991, pp. 140-149.
- [J.K. 94] J. Karlsson, P. Lidén, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms", *IEEE Micro*, vol. 14, no. 1, Feb. 1994, pp. 8-23.
- [J.L. 88] Jean-Claude Laprie, "Sûreté de fonctionnement et tolérance aux fautes: Concepts de base", *Rapport LAAS n° 88.287*, Novembre 1988.
- [J.M. 94] J.S. Melinger, S. Buchner, D. McMorro, W.J. Stapor, T.R. Weatherford, A.B. Campbell, H. Eisen, "Critical Evaluation of the Pulsed Laser Method for Single Event Effects Testing and Fundamental Studies", *IEEE Trans. On Nucl. Sci.*, vol. 41, n°6, Dec. 1994, pp. 2574-2584.
- [J.O. 92] J. Ohlsson, M. Rimen, U. Gunneflo, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog", in *IEEE International Symposium on Fault Tolerant Computing (FTCS-22)*, July 92, pp. 316-325.
- [J.P. 83] J.K. Pickel, "Single Events Upsets Mechanisms and Predictions", *Tutorial Short Course, IEEE NSREC'83*, 1983.
- [J.V. 98] J.M. Voas and G. McGraw, "Software Fault Injection: Inoculating Programs against Errors", *John Wiley and Sons*, New York, 1998.
- [J.Z. 87] J. A. Zoutendik, L. S. Smith, G. A. Soli, and R. Y. Lo, "Experimental evidence for a new SEU model in a CMOS SRAM obtained from model verification", *IEEE Trans. Nucl. Sci.*, vol. 34, Dec. 1987, pp. 358-366.

- [K.C. 95] K. T. Cheng, S.Y. Huang, and W. J. Dai, "Fault emulation: a new approach to fault grading", IEEE-ACM International Conference on Computer-Aided Design, November 1995, pp. 681-686.
- [K.C. 99] K. T. Cheng, S.Y. Huang, and W. J. Dai, "Fault emulation: A new methodology for fault grading", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, October 1999, 18(10):1487-1495.
- [L.A. 00] L. Antoni, R. Leveugle, B. Fehér, "Using Run-time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 405-413.
- [L.A. 03-a] L. Antoni, "Injection de Fautes par Reconfiguration Dynamique de Réseaux Programmables", Thèse de l'Institut National Polytechnique de Grenoble (INPG), spécialité Microélectronique, soutenue le 19 Septembre 2003 à Grenoble.
- [L.A. 03-b] L. Antoni, R. Leveugle, B. Fehér, "Using run-time reconfiguration for fault injection applications", IEEE Transactions on Instrumentation and Measurement Technology Conference, October 2003, pp. 1468- 1473.
- [L.B. 96] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape, "Serial fault emulation", In IEEE Design Automation Conference, June 1996, pp. 801-806.
- [L.B. 02] L. Berrojo, I. González, F. Corno, M. Sonza Reorda, G. Squillero, L. Entrena, C. López, "New Techniques for Speeding-Up Fault-Injection Campaigns", In Design, Automation and Test in Europe Conference and Exhibition (DATE'02), March 2002, pp. 847-852.
- [M.B. 97] M. BAZE, S. BUCHNER, "Attenuation of Single Event Induced Pulses in CMOS Combinational Logic", IEEE Trans. on Nuclear Science, Vol. 44, No 6, December 1997.
- [M.H. 97] M. C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools", IEEE Computer, vol. 30, no 4, April 1997, pp. 175-82.
- [M.L. 00] M. Lajolo, M. Rebaudengo, M. S. Reorda, M. Violante, and L. Lavagno, "Evaluating system dependability in a co-design framework", In IEEE Design, Automation and Test in Europe Conference and Exhibition, March 2000, pp. 586-590.
- [M.N. 99] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies", in proceedings of the 17th IEEE VLSI Test Symposium, April 26-30, 1999, p.86.
- [M.R. 92] M. Rimen and J. Ohlsson, "A study of the error behaviour of a 32-bit RISC subjected to simulated transient fault injection", in IEEE International Test Conference, Sept. 92, pp 696.
- [M.S. 99-a] M. B. Santos and J. P. Teixeira, "Defect-oriented mixed-level fault simulation of digital systems-on-a-chip using HDL", In IEEE-ACM Design, Automation and Test in Europe Conference and Exhibition", March 1999, pp. 549-553.

- [M.S. 99-b] M. B. Santos, F. M. Goncalves, I. C. Teixeira, J. P. Teixeira, "Defect-oriented Verilog fault simulation of soc macros using a stratified fault sampling technique", In IEEE VLSI Test Symposium", April 1999, pp. 326-332.
- [M.S. 03] M. Sonza-Reorda, M. Violante, "Accurate and efficient analysis of single event transients in VLSI circuits", 9th IEEE International On-Line Testing symposium, Kos, Greece, July 7-9, 2003, pp. 101-105.
- [N.K. 95] N.A. Kanawati, G.A. Kanawati, J.A. Abraham, "Dependability evaluation using hybrid fault/error injection", in IEEE International Computer Performance and Dependability Symposium, April 1995, pages 224-233.
- [N.T. 97] N. A. Toubia, E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection", IEEE transactions on Computer-Aided Design, vol. 16, no. 7, July 1997, pp. 783-789.
- [P.A. 00] P. Adell, R. D. Schrimpf, H. J. Barnaby, R. Marec, C. Chatry, P. Calvel, C. Barillot, and O. Mion, "Analysis of Single Event Transients in Analog Circuits", IEEE Trans. On Nucl. Sci., vol. 47, n°6, Dec. 2000, pp. 2616-2623.
- [P.C. 01-a] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante, "Exploiting circuit emulation for fast hardness evaluation", IEEE Transactions on Nuclear Science, July 2001, 48:2210-2216.
- [P.C. 01-b] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante, "Exploiting FPGA for accelerating fault injection experiments", in IEEE International On-Line Testing Workshop, July 2001, pages 9-13.
- [P.C. 01-c] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante, "Exploiting fpga-based techniques for fault injection campaigns on VLSI circuits", in IEEE International Symposium on Defect and fault Tolerance in VLSI Systems, October 2001, pages 250-258.
- [P.C. 01-d] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante, "Fpga-based fault injection for microprocessor systems", in Asian Test Symposium, November 2001, pages 304-309.
- [P.C. 03] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, M. Violante, "New techniques for efficiently assessing reliability of SOCs", Microelectronics Journal, Vol. 34, No. 1, January 2003, pp. 53-61.
- [P.D. 96] P. Dodd et al., "Impact of technology trends on SEU in CMOS SRAMs", IEEE Trans. Nucl. Sci., vol. 43, Dec. 1996, pp. 2797-2804.
- [P.H. 00] P. Hazucha, "Background Radiation and Soft Errors in CMOS Circuits", doctoral dissertation, no. 638, Linköping Univ., Sweden, 2000.

- [P.R. 99] P. Roche, J. M. Palau, G. Bruguier, C. Tavernier, R. Ecoffet, and J. Gasiot, "Determination of key parameters for SEU occurrence using 3-D full cell SRAM simulations", IEEE Trans. Nucl. Sci., vol. 46, 1999, pp. 1354–1362.
- [P.S. 01] P. Sundararajan and S. A. Guccione, "Run-time defect tolerance using jbits", in ACM International Symposium on Field Programmable Gate Arrays, ACM Press, 2001, pages 193-198.
- [P.S. 02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic", proc. Int'l Conf. Dependable Systems and Networks (DSN-2002), 2002, pp. 389-398.
- [R.J. 94] R. Johansson, "On Single Event Upset Error Manifestation", proc. First European Dependable Computing Conf. (EDCC-1), 1994, pp. 217-231.
- [R.K. 00] R. Koga, S.H. Crain, K.B. Crawford, S.C. Moss, S.D. Lalumondiere, J.W. Howard, "Single Event Transient (SET) Sensitivity of Radiation Hardened and COTS Voltage Comparators", IEEE NSREC 2000 Data Workshop Proceedings, 2000, pp. 53-60.
- [R.L. 99] R. Leveugle, "Towards modelling for dependability of complex integrated circuits", 5th IEEE International On-Line Testing workshop, Rhodes, Greece, July 5-7, 1999, pp. 194-198.
- [R.L. 00] R. Leveugle and K. Hadjiat, "Optimized generation of vhdl mutants for injection of transition errors", in 13th Symposium on Integrated Circuits and Systems Design (SBCCI2000), Manaus, Brazil, Sept. 18-24, 2000, pp. 243-248.
- [R.L. 01] R. Leveugle, "A Low-Cost Hardware Approach to Dependability Validation of IPs", in The IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, San Francisco, California, USA. Oct. 24-26, 2001, Los Alamitos, CA: IEEE Computer Society Press, 2001, pp. 242-249.
- [R.L. 04] R. Leveugle, "Early analysis of fault-attack effects for cryptographic hardware", Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), Florence, Italy, June 30, 2004, in Supplemental volume of the 2004 International Conference on Dependable Systems and Networks (DSN), 2004, pp. 348-353.
- [R.L. 05] R. Leveugle, "A New Approach for Early Dependability Evaluation Based on Formal Property Checking and Controlled Mutations", 11th IEEE International On-Line Testing Symposium, Saint Raphael, French Riviera, France, July 6-8, 2005.
- [R.M. 99] R.J. Martínez, P.J. Gil, G. Martín, C. Pérez, and J.J. Serrano, "Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection", Dependable Computing for Critical Applications (Proc. Seventh IFIP Working Conf. Dependable Computing for Critical Applications: DCCA-7), C.B. Weinstock and J. Rushby, eds., Jan. 1999, pp. 249-265.

- [R.M. 01] R. Marec, C. Chatry, P. Adell, O. Mion, C. Barillot, P. Calvel, L. Cresciucci, "Towards a Single Event Transient Hardness Assurance Methodology", RADECS 2001 Proceedings, 2001, pp. 343-350.
- [R.P. 02] R. Pease, A. Sternberg, Y. Boulghassoul, L. Massengill, S. Buchner, D. McMorrow, D. Walsh, G. Hash, S. Lalumondiere, and S. Moss, "Comparison of SETs in Bipolar Linear Circuits Generated with an Ion Microbeam, Laser Light and Circuit Simulation", IEEE Trans. On Nucl. Sci., vol. 49, n°6, Dec. 2002.
- [R.S. 97] R. Sedaghat-Maman and E. Barke, "A new approach to fault emulation", In IEEE International Workshop on Rapid System Prototyping, June 1997, pp. 173-179.
- [R.S. 98-a] R. Sedaghat-Maman and E. Barke, "Real time fault injection using logic emulators", In IEEE Asia and South Pacific Design Automation Conference, Feb. 1998, pp. 475-479.
- [R.S. 98-b] R. Sedaghat-Maman, "Fault emulation with optimized assignment of circuit nodes to fault injectors", In IEEE International Symposium on Circuits and Systems, June 1998, vol. 6, pp. 135-138.
- [R.V. 98] R. Velazco, P. Cheynet, A. Bofill, R. Ecoffet, "THESIC : A testbed suitable for the qualification of integrated circuits devoted to operate in harsh environment", IEEE European Test Workshop, 1998, pp. 89-90.
- [R.V. 00-a] R. Velazco, S. Rezgui, R. Ecoffet, "Injecting C.E.U. (Code Emulating Upsets) to evaluate the error rate of microprocessor-embedded digital applications", 2000 Single Event Effects (SEE) Symposium, Manhattan Beach, Los Angeles, 11-13 April 2000.
- [R.V. 00-b] R. Velazco, S. Rezgui, R. Ecoffet, "Predicting Error Rate for Microprocessor-Based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection", IEEE Transactions on Nuclear Science, vol. 47, n°6, December 2000, pp. 2405-2236.
- [R.W. 94] R.W. Wieler, Z. Zhang, and R. D. McLeod, "Using an FPGA based computer as a hardware emulator for built-in self-test structures", In IEEE International Workshop on Rapid System Prototyping, June 1994, pp. 16-21.
- [R.W. 95] R.W. Wieler, Z. Zhang, and R. D. McLeod, "Emulating static faults using a Xilinx based emulator", In IEEE Symposium on FPGAs for Custom Computing Machines, April 1995, pp. 110-115.
- [S.B. 96] S. Buchner, D. McMorrow, J. Melinger, A.B. Campbell, "Laboratory Tests for Single Event Effects", IEEE Trans. On Nucl. Sci., vol. 43, n°2, Apr. 1996, pp. 678-686.
- [S.B. 01] S. Buchner, D. McMorrow, A. Sternberg, L. Massengill, R.L. Pease, M. Maher, "Single Event Transient (SET) Characterization of a LM119 Voltage Comparator: An Approach to SET Model Validation Using a Pulsed Laser", RADECS 2001, IEEE Trans. On Nucl. Sci., vol. 49, n°3, 2002, pp. 1502-1508.

- [S.H. 93] S. Han, H. Rosenberg, and K. Shin, "DOCTOR: an Integrated Software Fault Injection Environment", Tech. Report CSE-TR-192-93, University of Michigan, 1993.
- [S.S. 97] S. Svensson and J. Karlsson, "Dependability Evaluation of the THOR Microprocessor Using Simulation-Based Fault Injection", Technical Report No. 295, Chalmers University of Technology, Department of Computer Engineering, Sweden, November 1997.
- [T.D. 96] T. A. DeLong, B. W. Johnson, and J. A. Profeta, "A fault injection technique for VHDL behavioural-level models", IEEE Design and Test of Computers, Winter 1996, 13:24-33.
- [T.M. 94] T. Michel et al., "Taking advantage of ASICs to improve dependability with very low overheads", ED&TC, 1994, pp. 14-18.
- [U.G. 89] U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", in Proc. 19th Int. Symp. FaultTolerant Computing (FTCS-19), Chicago, MI, USA, IEEE Computer Society Press, 1989, pp.340-7.
- [V.P. 99] V. Pouget, D. Lewis, H. Lapuyade, R. Briand, P. Fouillat, L. Sarger and M.-C. Calvet, "Validation of radiation hardened designs by pulsed laser testing and SPICE analysis", Microelectron. Reliab., vol. 39, 1999, pp. 931–935.
- [V.S. 97] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions", in Proc. 27th Int. Symp. on Fault-Tolerant Computing (FTCS-27), Seattle, WA, USA, June 1997, pp. 32-36.
- [Y.C. 98] Y. Crouzet, P. Thévenod-Fosse, and H. Waeselynck, "Validation of Software Testing by Fault Injection: The SESAME Tool", Proc. 11th Conf. Reliability and Maintainability, 1998, pp. 551-559.
- [Y.M. 95] Y. Moreau, S. Duzellier, and J. Gasiot, "Evaluation of the upset risk in CMOS SRAM through full three dimensional simulation", IEEE Trans. Nucl. Sci., vol. 42, Dec. 1995, pp. 1789–1796.
- [Z.S. 88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT - Fault Injection Based Automated Testing Environment", in Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18), 1988, pp. 102-107.

Publications

- **Revue internationale**

[R.L. 03] R. LEVEUGLE, K. HADJIAT, "Multi-level fault injections in VHDL descriptions: alternative approaches and experiments", *Journal of Electronic Testing: Theory and Application (JETTA)*, Vol. 19, No. 5, October 2003, pp. 559-575.

[A.A. 05] A. AMMARI, K. HADJIAT, R. LEVEUGLE, "Combined Fault Classification and Error Propagation Analysis to Refine RT-Level Dependability Evaluation", *Journal of Electronic Testing: Theory and Application (JETTA)*, à paraître en Août 2005.

- **Conférences internationales et Workshops**

[R.L. 00] R. LEVEUGLE, K. HADJIAT, "Optimized generation of VHDL mutants for injection of transition errors", 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00), Manaus, Amazonas, Brazil, September 18-24, 2000, pp. 243-248.

[R.L. 01] R. LEVEUGLE, K. HADJIAT, "Early prediction of SEU consequences through VHDL mutant generation", 6th European Conference on Radiation and its Effects on Components and Systems (RADECS 01), Grenoble, France, September 10-14, 2001 (oral presentation).

[R.L. 02] R. LEVEUGLE, K. HADJIAT, "Multi-level fault Injection experiments based on VHDL descriptions: a case study", 8th IEEE International On-Line Testing Workshop (IOLT'02), Bendor, France, July 8-10, 2002, pp. 107-111.

[A.A. 04-b] A. AMMARI, K. HADJIAT, R. LEVEUGLE, "On Combining Fault Classification and Error Propagation Analysis in RT-Level Dependability Evaluation", 10th IEEE International On-Line Testing Symposium (IOLTS 04), Madeira Island, Portugal, July 12-14, 2004.

[K.H. 05-a] K. HADJIAT, R. LEVEUGLE, "Early Dependability Evaluation: Injection of Multiple Bit-Flips", 6th IEEE Latin-American TestWorkshop (LATW 05), Brazil, Salvador, Bahia, March 30th - April 2nd, 2005.

[K.H. 05-b] K. HADJIAT, A. AMMARI, R. LEVEUGLE, "Early Functional Evaluation of SET Effects", 8th European Conference On Radiation and Its Effects on Components and Systems (RADECS 05), Palais des Congrès du Cap d'Agde, France, Sept. 19-23, 2005.

- **Conférences nationales**

[K.H. 02-a] K. HADJIAT, R. LEVEUGLE, "Évaluation prédictive de la sûreté de fonctionnement d'un circuit intégré", Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM), Grenoble, France, April 23-25, 2002, pp. 233-234.

[K.H. 02-b] K. HADJIAT, R. LEVEUGLE, "Expériences d'injection de fautes multi-niveaux dans des descriptions VHDL", Colloque CAO de circuits et systèmes intégrés, Paris, France, May 15-17, 2002, pp. 51-54.

[K.H. 04] K. HADJIAT, A. AMMARI, R. LEVEUGLE, "Application et Combinaison de Deux Approches d'Analyse de Sûreté", Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM), Marseille, France, May 4-6, 2004, pp. 410-412.

[K.H. 05-c] K. HADJIAT, A. AMMARI, R. LEVEUGLE, "Modélisation de fautes et génération de mutants pour analyse de robustesse de circuits sécurisés", Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM), Paris, France, May 10-12, 2005.

RESUME

La probabilité des fautes transitoires augmente avec l'évolution des technologies. Ceci a suscité un intérêt croissant pour les méthodes prédictives d'analyse des comportements erronés d'un circuit. Ce travail porte sur l'étude de deux aspects complémentaires : l'injection de fautes dans des circuits décrits au niveau RTL et l'analyse des résultats obtenus à l'issue de campagnes d'injection.

Nous présentons une nouvelle approche pour la génération de mutants, permettant l'instrumentation d'un circuit pour des modèles de fautes hétérogènes. Pendant la définition d'une campagne d'injection de fautes, le flot d'analyse que nous avons proposé permet au concepteur d'introduire, dans le même circuit, des inversions de bits uniques (SEU) ou multiples (MBF), ou encore des transitions erronées. En outre, nous avons visé une génération de mutant la plus efficace selon plusieurs contraintes qui incluent (1) la modification simple et automatique de la description initiale du circuit, (2) l'optimisation des entrées additionnelles pour le contrôle d'injection et (3) la réduction du surcoût matériel après synthèse pour une bonne compatibilité avec des campagnes d'injection de fautes basées sur l'émulation.

Dans le flot d'analyse, un modèle comportemental est généré permettant au concepteur d'identifier les chemins de propagation d'erreurs dans le circuit. Une telle analyse vise à identifier, très tôt dans le flot de conception, les modes de défaillance inacceptables d'un circuit afin de modifier immédiatement sa description et ainsi améliorer sa robustesse.

Nous présentons des résultats obtenus suite à des injections multi-niveaux dans des descriptions VHDL de circuits numériques. Ces résultats démontrent qu'une campagne d'injection réalisée très tôt dans le processus de conception, sur une description encore très éloignée de l'implémentation finale, peut donner des informations très utiles sur les caractéristiques de sûreté d'un circuit.

MOTS-CLES

Circuits numériques, analyse de sûreté, injection de fautes, mutants VHDL RTL, SEU, inversion de bits unique, inversion de bits multiple.

TITLE

EARLY PREDICTION OF DEPENDABILITY OF COMPLEX DIGITAL CIRCUITS

ABSTRACT

The probability of transient faults increases with the evolution of technologies. There is a corresponding increased demand for an early analysis of erroneous behaviours. This work concerns the study of two complementary aspects: fault injection in circuits described in RT-level VHDL and analysis of the results obtained at the end of a fault injection campaign.

We present a new approach for mutant generation, allowing circuit instrumentation for heterogeneous fault models. The proposed analysis flow allows a designer to mix single bit-flips (SEU), multiple bit-flips and erroneous transitions when defining a fault injection campaign. Furthermore, we target the most efficient generation with respect to multiple constraints, including (1) simple and automatic modification of the initial circuit description, (2) limited set of additional inputs for injection control and (3) limited hardware overhead after synthesis for compatibility with emulation-based fault injection campaigns.

In the analysis flow, a behavioural model is generated, allowing the designer to identify the detailed error propagation paths in the circuit. Such an analysis aims at identifying very early in the design flow the unacceptable failure modes of a circuit, in order to immediately modify its description to improve its robustness.

We report on results obtained with multi-level fault injections in VHDL descriptions of digital circuits. These results show that an analysis performed at a very early stage in the design process can actually give a helpful insight into the response of a circuit when a fault occurs.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-044-X (brochée)

ISBN : 2-84813-045-8 (électronique)