Analyse de robustesse de systèmes intégrés numériques

Kais Chibani

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nanoélectronique et Nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

Kais CHIBANI

Thèse dirigée par **Régis LEVEUGLE**
Co-encadrée par **Michele PORTOLAN**

préparée au sein du **Laboratoire TIMA**
dans **l'École Doctorale EEATS**

Analyse de robustesse de systèmes intégrés numériques

Thèse soutenue publiquement le **10 novembre 2016**,
devant le jury composé de :

M. Jean-Michel Portal
Professeur, Université d'Aix-Marseille, Président
M. Matteo Sonza-Reorda
Professeur, Politecnico de Turin, Rapporteur
M. Giorgio Di Natale
Directeur de Recherche CNRS, Laboratoire LIRMM, Rapporteur
M. Régis Leveugle
Professeur, Université Grenoble Alpes, Grenoble-INP, Directeur de thèse
M. Michele Portolan
Maître de conférences, Université Grenoble Alpes, Grenoble-INP, Co-encadrant
M. Florent Miller
Responsable d'équipe de Recherche, Airbus Group Innovations, Examinateur

Dédicace

A ma mère, A mon père

Ce manuscrit de thèse représente le fruit de vos années d'investissement. Je suis fier de vous le dédier, veuillez y trouver le témoignage de ma grande reconnaissance. Merci d'avoir contribué à ma réussite. Je vous souhaite la bonne santé et que dieu vous garde.

A Hela

Merci d'être là ! Aucun mot ne pourra jamais suffire pour dire tout ce que j'ai dans mon cœur, bien au-delà de ces pages. Que dieu réunisse nos chemins pour un long commun serein et que ce travail soit le témoignage de ma reconnaissance et de mon amour sincère et fidèle.

A Khalil & Nihel, Seif & Naouel

Je souhaite un avenir radieux et plein de gloire pour vous et pour vos enfants. Je vous dédie ce travail en témoignage de ma grande affection.

A la mémoire de mon très cher oncle Hédi.

Remerciements

Ces travaux de thèse ont été réalisés au sein du laboratoire TIMA de Grenoble, dont je tiens à remercier la directrice, Mme. Dominique Borrione, pour m'y avoir accueilli et M. Salvador Mir, qui a repris la direction, pour m'avoir conservé.

Je tiens à exprimer ma profonde gratitude aux membres du jury, en commençant par M. Jean-Michel Portal, Professeur à l'Université d'Aix-Marseille, en sa qualité de président de mon jury. Je tiens également à remercier les rapporteurs de ce jury, M. Matteo Sonza-Reorda, Professeur au Politecnico di Torino, et M. Giorgio Di Natale, Directeur de Recherche CNRS au Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) pour l'intérêt qu'ils ont porté à ce travail ainsi que pour leurs commentaires et suggestions. Un grand merci à M. Florent Miller, Responsable de l'équipe Semiconductor & Equipment Dependability à Airbus Group Innovations, qui fût mon examinateur lors de la soutenance.

Je tiens tout particulièrement à remercier mon directeur de thèse M. Régis Leveugle, Professeur à Grenoble INP et co-responsable de l'équipe AMfoRS au TIMA, pour m'avoir proposé un sujet de thèse passionnant. Je lui dois des pistes de recherche originales, ainsi qu'une formation scientifique et méthodologique de qualité. Toute ma reconnaissance pour son soutien, ses précieux conseils, ses critiques et pour la confiance qu'il m'a accordée d'abord en stage puis en thèse. J'ai eu l'honneur d'avoir été l'un de ses doctorants.

Je remercie sincèrement mon co-encadrant et mon collègue de bureau M. Michele Portolan, Maître de conférences à Grenoble INP, pour son encadrement, son dynamisme, ses encouragements et ses conseils qui m'ont permis de progresser et mener à bien ce travail. J'ai eu le plaisir d'avoir été son premier doctorant.

Cette thèse n'aurait pu être possible sans le support et la bonne humeur de plusieurs personnes qui, chacune de sa façon, m'ont aidé, soutenu ou simplement accompagné pendant ces trois années. C'est à eux que je dois des moments inoubliables, passés dans un cadre amical, encourageant et enrichissant. Mille mercis donc à Asma, Simon, Paolo, Amine, Mohamed & Salma, Alban, Pierre, Diego; sans oublier, Hamza & Rahma, Rania et mon cousin Chedly.

Je voudrais remercier l'ensemble du personnel administratif du laboratoire TIMA. Je pense notamment à Anne-Laure Fourneret-Itié et Laurence Ben Tito. J'adresse aussi mes remerciements aux administrateurs du CIME, Alexandre Chagoya et Robin Rolland pour leur aide dans le traitement des problèmes techniques et pour leur enthousiasme.

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

Enfin, je ne remercierais jamais assez mes chers parents, Mahmoud & Nejiba, mes deux sœurs, Nihel & Naouel, mes beaux-frères, Seif & Khalil et ma chère Halhoula 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

Remerciements ..... **I**
Table des matières ..... **III**
Liste des figures ..... **V**
Liste des tableaux ..... **VII**
Introduction générale **9**
Chapitre I : Sources de perturbations et effets sur les circuits numériques **13**
I.1. Introduction 13
I.2. Perturbations d'origine naturelle 13
I.2.1. Environnement radiatif spatial 13
I.2.2. Environnement radiatif atmosphérique 16
I.3. Perturbations d'origine intentionnelle 17
I.3.1. Perturbation de la tension d'alimentation 18
I.3.2. Perturbation de l'horloge 18
I.3.3. Perturbation de la température 18
I.3.4. Perturbations électromagnétiques 19
I.3.5. Perturbations optiques ou par particules 19
I.3.6. Résumé des perturbations intentionnelles 21
I.4. Effets des perturbations sur les circuits intégrés 22
I.4.1. Effets de dose 22
I.4.2. Les effets singuliers (SEE) 22
I.5. Modèles de fautes ou d'erreurs 25
I.6. Robustesse et sûreté de fonctionnement d'un système 27
I.7. Conclusion 29
Chapitre II : Etat de l'art sur les techniques d'évaluation de robustesse **31**
II.1. Introduction 31
II.2. Techniques d'injection de fautes 31
II.2.1. Injection de fautes par simulation 32
II.2.2. Injection de fautes par émulation 36
II.2.3. Comparaison des deux approches 39
II.3. Evaluation de robustesse des systèmes à base de microprocesseur 41
II.3.1. Analyse par mesure de l'AVF (Architectural Vulnerability Factor) 41
II.3.2. Analyse de robustesse par analyse du code 43
II.4. Conclusion 47
Chapitre III : Approche d'évaluation de robustesse des microprocesseurs fondée sur l'analyse de criticité des registres **49**
III.1. Introduction 49
III.2. Présentation générale de l'approche 49
III.3. Le véhicule d'étude : le processeur LEON3 51
III.3.1. L'architecture SPARC v8 51
III.3.2. Le processeur LEON3 53
III.3.3. Unité entière du LEON3 54

III.4. Mise en œuvre de l'approche : algorithme de prédiction de criticité 56
III.4.1. Méthode d'analyse basée sur une trace de simulation..... 57
III.4.2. Algorithme de calcul de criticité 59
III.5. Validation de l'approche par injection de fautes 68
III.5.1. Techniques d'injection de fautes utilisées..... 68
III.5.2. Analyse des résultats 71
III.6. Comparaison avec des évaluations de criticité obtenues lors de la compilation 75
III.6.1. Identification des registres critiques..... 75
III.6.2. Impact des options de compilation sur la criticité..... 76
III.6.3. Comparaison des durées des expérimentations et synthèse..... 78
III.7. Effets de quelques caractéristiques architecturales 79
III.7.1. Impact de la variation de la taille du banc de registres 80
III.7.2. Impact des différentes configurations de caches 81
III.8. Conclusion..... 82
Chapitre IV : Méthode d'analyse de durées de vie dans des circuits numériques 84
IV.1. Introduction 84
IV.2. Flot d'analyse de durées de vie 85
IV.2.1. Présentation générale 85
IV.2.2. Hypothèses 87
IV.3. Extraction des cônes logiques..... 88
IV.3.1. Cône logique..... 88
IV.3.2. Algorithme d'extraction 89
IV.4. Identification des signaux clés 95
IV.4.1. Notion de branche logique 95
IV.4.2. Validation de branche logique et identification des signaux clés 96
IV.5. Calcul de durées de vie 98
IV.5.1. Matrice d'impacts..... 98
IV.5.2. Algorithme de calcul de durées de vie 99
IV.6. Validation expérimentale..... 101
IV.6.1. Cohérence des résultats..... 102
IV.6.2. Durée des expérimentations..... 105
IV.7. Comparaison avec l'algorithme de prédiction de criticité des registres 106
IV.8. Conclusion..... 108
Conclusion générale et perspectives..... 111
Bibliographie..... 115
Publications de l'auteur..... 123
Glossaire..... 125
Annexes 127
A.1. Architecture SPARC v8 127
A.1.1. Formats des instructions 127
A.1.2. Instructions 128
A.1.3. Banc de registres et manipulation des fenêtres..... 129
A.2. Effets des caractéristiques architecturales sur la criticité 131

Liste des figures

Figure I-1 : Environnement radiatif spatial [WW. 1]. 14
Figure I-2 : Description des différentes composantes de l'environnement radiatif spatial. Variations des flux de particules en fonction de leur énergie [Boud. 99]. 16
Figure I-3 : Modification locale d'une période d'horloge. 18
Figure I-4 : Apparition de paires électron-trou lors d'un tir laser sur une jonction PN [Rosc. 13]. 21
Figure I-5 : Ionisation du matériau [Kay]. 23
Figure I-6 : Scénario de propagation d'une impulsion transitoire dans un circuit combinatoire (a) et les formes d'ondes des entrées des bascules (b). 26
Figure I-7 : Schéma Faute-Erreur-Défaillance. 28
Figure II-1 : Intervalles ACE et intervalles un-ACE. 43
Figure III-1 : Vue globale de l'approche d'analyse de criticité proposée. 51
Figure III-2 : Fenêtrage des registres dans l'architecture SPARC v8 [SPARC. 92]. 53
Figure III-3 : Schéma-bloc du processeur LEON3 [Gais. 13]. 54
Figure III-4 : Pipeline de l'unité entière du LEON3 [Gais. 13]. 55
Figure III-5 : Extrait du fichier généré après simulation du LEON3. 58
Figure III-6 : Algorithme de calcul de criticité des registres du LEON3. 59
Figure III-7 : Invalidation des instructions annulées dans la trace de simulation. 60
Figure III-8 : Détection d'un cycle de gel dans le pipeline. 61
Figure III-9 : Modèles associés aux instructions "ADD", "ADDimm" et "NOP". 63
Figure III-10 : Exemple de cas ou un registre source n'est pas critique. 64
Figure III-11 : Contenu du registre D_INST en cas d'instruction NOP. 67
Figure III-12 : Impact du mécanisme de *forwarding* sur l'utilisation d'un registre interne. 68
Figure III-13 : Flot d'une compagne d'injection par simulation. 69
Figure III-14 : Flot général détaillé d'injection par endo-reconfiguration [Benj. 13]. 70
Figure III-15 : Comparaison des résultats de criticité des registres du pipeline. 71
Figure III-16 : Comparaison des résultats de criticité des registres du banc de registres (CRC32). 72
Figure III-17 : Comparaison des distributions de criticité des registres internes selon la méthode d'évaluation (Injection/ Algorithme). 73
Figure III-18 : Registres internes critiques identifiés par l'algorithme et la technique d'injection. 74
Figure III-19 : Registres généraux critiques identifiés par l'algorithme et la technique d'injection (CRC32). 75
Figure III-20 : Registres critiques identifiés par l'algorithme et l'approche de compilation. 75
Figure III-21 : Nombre de lectures/écritures dans le banc de registres en fonction des macro-options pour l'application JPEG. 78
Figure III-22 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application JPEG. 80
Figure III-23 : Effets de la variation du nombre de sous-ensembles et de la taille des caches. 81
Figure IV-1 : Vue globale du flot d'analyse de durées de vie. 85
Figure IV-2 : Masquage logique d'une faute transitoire. 88
Figure IV-3 : Schéma d'un cône logique. 88
Figure IV-4 : Processus d'extraction des cônes à partir des données d'élaboration. 89
Figure IV-5 : Dissociation d'une instance de nature multiplexeur 2 vers 1. 90
Figure IV-6 : Exemple de graphe de causalité. 91
Figure IV-7 : Organigramme d'identification des cônes logiques. 93

Figure IV-8 : Exemple de reset synchrone (a) et de reset asynchrone (b) actifs au niveau haut. 94
Figure IV-9 : Identification des éléments du cône logique associé à la bascule B. 94
Figure IV-10 : Représentation d'un cône en fonction de branches logiques. 95
Figure IV-11 : Analyse des branches logiques du cône B. 96
Figure IV-12 : Algorithme d'identification des signaux clés. 97
Figure IV-13 : Branche logique liant la bascule A (feuille) à la bascule B (racine). 98
Figure IV-14 : Matrice d'impacts. 98
Figure IV-15 : Algorithme de calcul de durées de vie. 100
Figure IV-16 : Comparaison des résultats de criticité par bascule pour le circuit AES Parity. 102
Figure IV-17 : Comparaison des résultats de criticité par cycle pour le circuit AES Parity. 102
Figure IV-18 : Comparaison des résultats de criticité par bascule pour le circuit AES Morph. 103
Figure IV-19 : Comparaison des résultats de criticité par cycle pour le circuit AES Morph. 103
Figure IV-20 : Comparaison des résultats de criticité des bascules internes pour les applications CRC32, AES, FFT, JPEG, SHA, FIR et MTMX. 104
Figure IV-21 : Comparaison des résultats de criticité des registres internes donnés par les trois approches. 106
Figure IV-22 : Résultats de criticité des bits du registre d'instruction D_INST, obtenus avec l'analyse de durées de vie. 107
Figure A-1 : Format des instructions dans l'architecture SPARC V8 [SPARC. 92]. 127
Figure A-2 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application AES. 131
Figure A-3 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application FIR. 131
Figure A-4 : Effets de la variation du nombre de sous-ensembles et de la taille des caches, pour l'application AES. 132
Figure A-5 : Effets de la variation du nombre de sous-ensembles et de la taille des caches, pour l'application FIR. 132

Liste des tableaux

Tableau I-1 : Tableau de la représentation de particules dans l'espace [Boud. 95]. 15
Tableau I-2 : Comparaison de flux (cm-2. s-1) de particules [Gasio. 01]. 17
Tableau II-1 : Récapitulatif des avantages et des inconvénients des approches d'injection. 40
Tableau III-1 : Correspondance entre les noms logiques des registres et leurs numéros dans le banc de registres en fonction du CWP. 62
Tableau III-2 : Présentation des applications utilisées. 71
Tableau III-3 : Ordres de criticité en fonction des macro-options de compilation. 76
Tableau III-4 : Pourcentages de gel du pipeline des différentes versions de logiciel. 77
Tableau III-5 : Temps relatifs nécessaire pour chaque approche. 78
Tableau III-6 : Comparaison des trois approches d'évaluation de robustesse. 79
Tableau IV-1 : Caractéristiques des circuits analysés. 101
Tableau IV-2 : Comparaison des résultats de criticité globale. 104
Tableau IV-3: Temps moyen pris pour l'évaluation de criticité. 105
Tableau IV-4 : Evaluation du nombre de signaux suivis en simulation. 106
Tableau IV-5 : Gains des deux approches proposées par rapport à la technique d'injection de fautes par émulation. 108
Tableau IV-6 : Comparaison des quatre approches d'évaluation de robustesse. 108
Tableau A-1 : Les registres SPARC v8. 129

Introduction générale

Aujourd'hui, les évolutions technologiques dans le domaine de la microélectronique diminuent la taille des composants électroniques jusqu'aux dimensions nanométriques. Cette miniaturisation croissante a permis de réaliser des circuits présentant une très grande densité d'intégration, avec des performances toujours plus élevées. Mais elle a un impact très important sur la robustesse des systèmes numériques. En effet, les circuits électroniques sont de plus en plus sensibles vis-à-vis de leur environnement. Leurs dimensions plus petites et leurs tensions d'alimentation plus basses, entre autres, tendent à les rendre moins résistants à différents types de perturbations externes (rayonnements ou particules, variation de la température ou de l'alimentation …). Et surtout, malgré les efforts des fabricants, l'augmentation de la complexité des circuits augmente inexorablement la probabilité qu'une perturbation se traduise par une information erronée. Une perturbation peut engendrer des fautes appelées transitoires qui se traduisent par une modification de l'information contenue dans un élément de stockage, ou par un changement du résultat produit par une fonction logique. Ces fautes dites aussi "Soft-errors" (lorsque mémorisées) sont une altération des données traitées dans les circuits et en conséquence modifient le fonctionnement. Or le comportement erroné d'un circuit peut être tout à fait inacceptable surtout si celui-ci est mis en œuvre dans un système critique.

Un système est dit critique lorsque sa défaillance est susceptible d'entraîner des conséquences graves pour l'intégrité des personnes (calculateur de vol d'un avion), pour l'économie (chiffrement des transactions bancaires) ou pour l'environnement (contrôle d'un réacteur nucléaire). Outre l'industrie, les transports, la banque et l'énergie, on trouve également des systèmes critiques dans la santé (contrôle d'un pacemaker par exemple), le spatial (commandes d'un satellite qu'il sera impossible de réparer une fois sur orbite) ou la défense. Autant de contextes et conditions extrêmes où il faut se préparer à l'imprévisible et anticiper au maximum les situations afin de garantir le fonctionnement attendu.

Les concepteurs doivent par conséquent faire face à une réduction de la robustesse des circuits intégrés supportant des applications qui imposent des contraintes de sûreté et/ou de sécurité. La solution est d'implanter des mécanismes de protection face aux effets des perturbations environnementales. Les techniques classiques de protection mettent en œuvre du blindage, des mesures au niveau du processus de fabrication (technologie), des bibliothèques (fondeur) durcies, et des architectures redondantes. La contrepartie de cette solution est la difficulté de sa mise en place en cas de circuits complexes contenant des dizaines voire des centaines de millions de portes logiques. De plus, elle présente des limites en termes de performances et des coûts supplémentaires en surface, en puissance consommée et en temps de conception. Par exemple, les circuits intégrés "résistants aux radiations" sont coûteux et les technologies avec lesquelles ils sont fabriqués sont souvent moins avancées en terme de finesse des géométries.

Afin de se conformer aux exigences de fiabilité et d'optimiser en même temps les stratégies de protection, il est nécessaire d'évaluer les conséquences des erreurs potentielles et d'identifier les éléments les plus critiques. Ceci est dans le but de limiter les protections aux blocs les plus critiques, car conduisant à des dysfonctionnements inacceptables pour l'utilisateur en cas de perturbation. De ce fait, les concepteurs ont besoin d'outils et de méthodes efficaces pour étudier, analyser et comprendre les différentes vulnérabilités de leurs circuits lorsqu'une ou plusieurs erreurs surviennent pendant l'exécution. L'objectif final est la neutralisation de ces vulnérabilités et donc l'amélioration de la robustesse à moindre coût.

Pour des composants de stockage tels que les mémoires, qui sont conçues de façon aussi dense que possible, l'industrie a utilisé des codes correcteurs d'erreurs, désignés par l'acronyme ECC venant de l'expression anglo-saxonne "*Error Correcting Code*". Ce type de codage est capable de réduire les taux d'erreurs en sortie de plusieurs ordres de grandeur au prix de seulement quelques bits supplémentaires par mot. Pour la logique aléatoire (circuits de type calcul), malheureusement, il n'existe encore aucun remède aussi efficace, et toutes les solutions actuelles présentent des coûts bien supérieurs et/ou des limites en termes de performances. C'est la raison pour laquelle nous nous intéressons, dans le cadre de cette thèse, à cette partie des circuits. Pour les modules de calcul tels que les processeurs et les accélérateurs matériels, l'accroissement de la fréquence de fonctionnement du circuit et l'augmentation de leur complexité font qu'une perturbation dans le résultat d'une fonction logique a plus de chances d'être capturée au final par un élément de stockage (bascule, registre, mémoire).

Dans le cas d'un système à base de microprocesseur, le risque de dysfonctionnement dépend fortement de l'utilisation des registres internes par l'application considérée : plus la durée pendant laquelle une information stockée dans le registre peut être réutilisée est longue, plus le risque de dysfonctionnement en cas de perturbation est élevé. L'analyse de robustesse revient donc à évaluer précisément la criticité des différents registres en fonction des informations qu'ils contiennent pendant l'exécution de l'application. Ceci est compliqué, dans le cas de processeurs récents, par les nombreux registres internes implantés dans les pipelines et dont l'utilisation dépend fortement de l'application exécutée. Pour les blocs matériels spécialisés, l'utilisation effective des registres internes dépend également de l'application en cours d'exécution, et la robustesse de l'application en découle. Dans les deux cas, le critère clé est l'évaluation de la durée de vie des données stockées dans les registres, pour une application donnée. Au niveau de l'état de l'art, peu d'études ont été publiées sur ce thème. Lorsqu'il s'agit d'un processeur, les évaluations sont réalisées en faisant abstraction de l'architecture réelle du circuit, ou concernent les durées de vie des informations stockées en mémoire (notamment mémoires caches). Pour les blocs matériels spécifiques, aucune approche analytique n'a été validée.

Les travaux réalisés pendant cette thèse visent à proposer et développer des approches permettant d'analyser, tôt dans le flot de conception, la robustesse d'un système numérique d'une façon générale. Le premier objectif est de modéliser l'impact des architectures pipelines de microprocesseurs, afin de raffiner les évaluations de criticité en tenant compte des registres internes du pipeline. Le second objectif est de réaliser des analyses de durées de vie sur les

registres des blocs matériels spécifiques (blocs IP synthétisables) qu'ils soient de type microprocesseur ou non.

Ce manuscrit est divisé en quatre chapitres. Le chapitre 1 présente les différentes sources de perturbations environnementales (naturelles ou intentionnelles) et la problématique liée aux effets de ces perturbations sur les composants électroniques. L'analyse des phénomènes mis en jeux dans les circuits intégrés permet ensuite d'établir une liste de modèles de fautes. Nous étudions ainsi l'impact des fautes transitoire sur la sûreté de fonctionnement d'un système numérique.

Le chapitre 2 est consacré à une présentation de l'état de l'art des techniques d'évaluation de robustesse d'un circuit numérique durant la phase de conception. Celui-ci nous permettra de faire le point sur les avantages et les inconvénients des techniques existantes et nous conduira à proposer d'autres méthodes d'évaluation.

Le chapitre 3 propose une nouvelle approche qui tend à évaluer la robustesse d'un circuit à base de microprocesseur. L'évaluation repose sur l'analyse de criticité des registres implémentés dans la microarchitecture. Au début, nous présentons les caractéristiques et les spécificités de l'architecture du processeur LEON3 qui est notre cas d'étude. Puis, nous présentons la technique proposée pour le calcul de criticité et l'algorithme de prédiction de criticité qui a été élaboré. Ensuite, pour l'étape de validation, les résultats obtenus sont comparés avec ceux issus de plusieurs campagnes d'injection de fautes. La dernière partie de ce chapitre sera consacrée à une étude de l'effet de plusieurs caractéristiques architecturales sur la criticité globale d'une application. Ces évaluations mettront également en lumière les avantages les plus marquants de la nouvelle technique par rapport aux autres approches.

Le chapitre 4 présente enfin une méthodologie unifiée pour analyser la robustesse intrinsèque des circuits numériques. En effet, une nouvelle méthode d'analyse de durées de vie a été développée. Cette dernière permet d'estimer la criticité de toutes les bascules en calculant leur durée de vie pour une application donnée. Une première version d'un outil logiciel de mise en œuvre de cette approche a été réalisée afin de valider la méthode proposée. Dans un premier temps, l'outil a été appliqué sur des exemples de circuits purement matériels comme des crypto-processeurs de chiffrement. Ensuite, il a été testé sur le processeur LEON3 exécutant des applications variées. Pour les différents cas d'étude, les résultats fournis par l'outil sont comparés à des injections de fautes. La dernière partie de ce chapitre portera sur une comparaison avec l'approche décrite dans le chapitre 3. Ceci nous permettra de faire le point sur les techniques développées au cours de cette thèse.

Chapitre I : Sources de perturbations et effets sur les circuits numériques

I.1. Introduction

Le mauvais fonctionnement d'un circuit numérique peut être la conséquence de problèmes liés soit à sa conception et fabrication, par exemple quand celles-ci ne sont pas conformes à la spécification, soit de son vieillissement (dégradation des caractéristiques) ou à des perturbations de l'environnement. Dans le cadre de cette thèse nous nous intéressons à la dernière source de dysfonctionnement relative aux perturbations qui menacent potentiellement la fiabilité des composants électroniques. En effet, la réduction drastique des dimensions des transistors, mais aussi les tensions d'alimentation de plus en plus basses et l'accroissement des fréquences de fonctionnement, qui accompagnent l'évolution technologique vers les dimensions nanométriques, renforcent la sensibilité des circuits intégrés aux perturbations extérieures. Dans ce chapitre, nous distinguons deux types de perturbations : celles d'origine naturelle telles que les environnements radiatifs et celles d'origine intentionnelle visant à déstabiliser la sécurité des systèmes numériques. Nous étudions ensuite les effets de ces perturbations sur les circuits intégrés. Des modèles de fautes qui représentent ces effets seront présentés afin d'être utilisés dans les chapitres suivants pour analyser la robustesse des circuits.

I.2. Perturbations d'origine naturelle

Cette section a pour but de présenter l'environnement radiatif spatial et atmosphérique dans lequel opèrent les circuits électroniques. Une radiation est une énergie rayonnée sous forme d'ondes ou de particules. Ces particules deviennent dangereuses pour l'électronique dès lors que l'énergie qu'elles apportent est suffisante pour perturber directement ou indirectement le fonctionnement des systèmes. Une distinction est faite entre l'environnement radiatif spatial et l'environnement radiatif atmosphérique. En effet, la nature des particules entrant en jeu est différente dans les deux cas.

I.2.1. Environnement radiatif spatial

Le domaine spatial est très complexe vis-à-vis des particules qui s'y trouvent. Les circuits électroniques sont soumis à un environnement radiatif extrêmement contraignant. Il existe principalement quatre sources de rayonnement qui sont successivement le vent solaire, les éruptions solaires, le rayonnement cosmique et les ceintures de radiations (figure I-1). Les composants plongés dans cet environnement sont soumis à des particules d'origines et d'énergies diverses telles que des photons, des électrons, des protons et des ions couvrant une large gamme de numéros atomiques [Boud. 95].

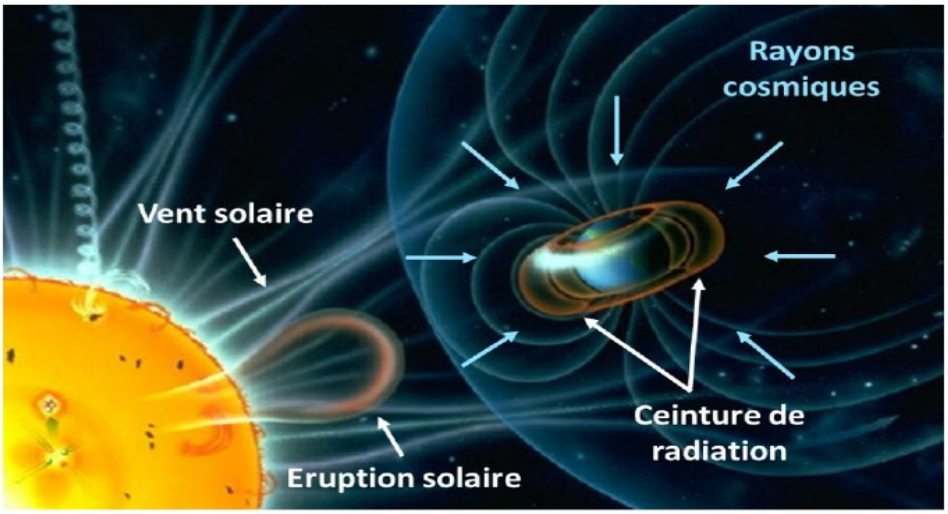


Figure I-1 : Environnement radiatif spatial [WW. 1].

Le vent solaire est un gaz ionisé peu dense résultant de l'évaporation de la couronne solaire. Il est constitué pour l'essentiel d'électrons, de protons et d'atomes d'hélium d'énergies inférieures à 100 keV. Vu leur faible énergie, ces particules sont très rapidement arrêtées et n'arrivent pas au contact des composants. Le vent solaire, en frappant la Terre (figure I-1), confine son champ magnétique dans une zone appelée magnétosphère, qui est une déformation ovoïdale du champ dipolaire, puis s'écoule le long de l'onde de choc.

Les éruptions solaires sont les événements les plus énergétiques se manifestant dans le système solaire. Elles se produisent lorsque l'énergie piégée dans les champs magnétiques associés aux taches solaires est brutalement libérée. Les éruptions engendrent des émissions électromagnétiques dans toutes les longueurs d'ondes, des ondes radio aux rayons X et gamma, et propulsent des particules à haute énergie dans l'espace, lesquelles peuvent mettre de 2 à 4 jours pour parvenir au niveau de l'orbite de la Terre. Les éruptions solaires sont de 2 types : les éruptions solaires à protons avec un spectre d'énergie assez important pouvant atteindre jusqu'à une centaine de MeV et les éruptions à ions lourds dont l'énergie varie d'une dizaine a une centaine de MeV.

Le rayonnement cosmique est constitué de particules énergétiques, par exemple des protons et des ions d'hélium, qui se déplacent dans l'espace intergalactique. Ces particules proviennent de phénomènes qui surviennent au-delà de notre système solaire. Lorsqu'elles pénètrent dans l'atmosphère terrestre, elles entrent en collision avec les atomes de notre atmosphère, et les brisent, ce qui produit un rayonnement secondaire, d'intensité moindre. Il faut savoir qu'une particule provenant de l'espace a une vitesse d'au moins 200 000 km/s, à savoir deux tiers de la vitesse de la lumière mais les rayons cosmiques ne parviennent pas directement au sol de la Terre car ils entrent en collision avec les atomes de la haute atmosphère. Ils engendrent ainsi de nombreuses particules secondaires : des protons, neutrons et électrons. Les particules cosmiques primaires sont des atomes privés de leurs électrons à cause des températures extrêmes présentes dans des phénomènes conséquents à l'explosion d'une étoile d'où ils sont issus. Il existe différents types de particules dans ces rayons cosmiques. La plupart sont des noyaux d'hydrogène mais ils peuvent aussi être composés d'autres noyaux :

85% de noyaux d'hydrogène, environ 12% de noyaux d'hélium, le reste étant constitué d'électrons et de différents nucléons résultant des rayons alpha et bêta.

Les ceintures de radiations, encore appelées ceintures de Van Allen, tirent leur nom du physicien américain James Alfred Van Allen qui les a découvertes en 1958, grâce au premier satellite américain Explorer 1. Ce sont des régions de la magnétosphère dans lesquelles sont piégées des particules chargées très énergétiques issues du vent solaire et des rayons cosmiques. L'énergie de ces particules dépasse couramment le MeV, c'est à dire l'énergie qu'aurait un électron accéléré par un champ électrique d'un million de Volts. Le piégeage de ces particules est dû au champ magnétique terrestre qui, sans ralentir les particules, les force à décrire des boucles autour de la Terre. Les particules, qui mettraient une fraction de seconde à traverser l'environnement terrestre si elles allaient en ligne droite, peuvent rester emprisonnées dans le champ magnétique terrestre pendant plusieurs semaines. Il existe principalement trois ceintures de Van Allen. Deux de ces ceintures formées d'électrons sont situées à 9000km et 30000km. La troisième ceinture, formée principalement de protons, se trouve à 12000km [Boud. 99]. Les protons ont des énergies allant de 100KeV à plusieurs centaines de MeV. Les électrons quant à eux ont des énergies comprises entre quelques dizaines d'eV et 7MeV. Ces particules chargées sont injectées en grande partie par la queue de la magnétosphère. Les particules énergétiques qui évoluent dans les ceintures de radiations y adoptent des trajectoires qui suivent les lignes de force du champ magnétique terrestre.

Le tableau I-1 résume la nature et la provenance des particules rencontrées dans l'environnement spatial ainsi que leurs énergies et flux.

Tableau I-1 : Tableau de la représentation de particules dans l'espace [Boud. 95].

PROVENANCE	PARTICULES	ENERGIES	FLUX
Ceintures de radiations	Protons Electrons	< qq 100 MeV (dont 99%<10 MeV) <7 MeV (dont 99%<2 MeV)	10 à 10^6 cm-²s-1 10^{-2} à 10^7 cm-²s-1
Vent solaire	Protons Electrons Particules α (7 à 8%)	<100 KeV <qq KeV	10^8 à 10^{10} cm-²s-1
Eruptions solaires	Protons Particules α Ions lourds	10 MeV à 1 GeV 10 MeV à qq 100 MeV	10^{10} cm-²s-1 ~ 10^2 à 10^3 cm-²s-1
Rayons cosmiques	Protons (87%) Particules α (12%) Ions lourds (1%)	10^2 à 10^6 MeV Fortes énergies 1 MeV à 10^{14} MeV	1 cm-²s-1 100 MeV 10^{-4} cm-²s-1 10^6 MeV

D'après la figure I-2, nous pouvons distinguer trois types de radiations :

- Les particules à fort débit mais de faible énergie et donc faciles à arrêter,
- Les particules qui ont un débit et une énergie intermédiaires,

- Les particules de très forte énergie mais dont le flux très faible implique une interaction peu probable.

Les particules correspondant à la zone intermédiaire sont, bien entendu, les plus contraignantes pour l'électronique car difficiles à arrêter et importantes en nombre.

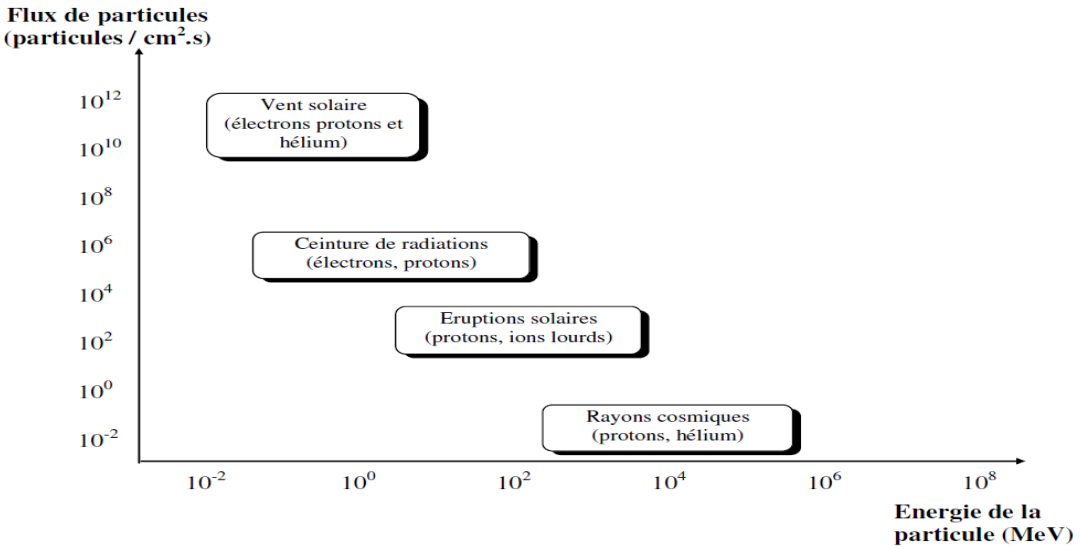


Figure I-2 : Description des différentes composantes de l'environnement radiatif spatial. Variations des flux de particules en fonction de leur énergie [Boud. 99].

I.2.2. Environnement radiatif atmosphérique

La terre est protégée par l'atmosphère qui constitue un véritable écran semi-perméable laissant passer la lumière mais arrêtant la plus grande partie des radiations issues de l'espace. L'environnement radiatif atmosphérique est donc beaucoup moins agressif que l'environnement spatial mais n'en reste pas moins potentiellement dangereux. Il est principalement dû à l'interaction du rayonnement cosmique avec les atomes de l'atmosphère [Vial. 98]. En effet, à l'entrée des rayons cosmiques dans l'atmosphère, on assiste à une atténuation des niveaux d'énergie ainsi qu'à la désintégration de ces rayons en différentes particules. Ceci est causé notamment par l'interaction avec les atomes d'azote et d'oxygène [Bolch. 10].

Les particules hautement énergétiques provenant des rayonnements cosmiques ne sont pas arrêtées par le champ magnétique terrestre. Elles entrent alors en collision avec les atomes constituant l'atmosphère et interagissent de deux façons différentes. Dans la première, en ionisant directement les éléments de l'atmosphère elles perdent une partie de leur énergie. Dans la deuxième, elles déclenchent sur ces éléments des réactions nucléaires en chaîne formant ainsi une cascade de particules secondaires telles que des protons, électrons, neutrons, ions lourds, muons et pions. Ces particules peuvent à leur tour interagir avec les molécules de l'atmosphère. Du fait de ces collisions multiples et de la perte d'énergie consécutive, le flux de radiations diminue à mesure qu'il se rapproche du sol car à l'issue de chaque interaction, une perte d'énergie importante a lieu. Par ailleurs, il varie aussi en latitude et il est 4 fois plus important aux pôles qu'à l'équateur.

Le tableau I-2 présente les flux de particules secondaires à deux altitudes caractéristiques : la surface de la Terre et l'altitude moyenne des vols commerciaux. Les valeurs sont tirées de [Gasio. 01]. Elles montrent en particulier qu'au niveau des altitudes avioniques, les particules prédominantes sont les neutrons et les électrons à haute énergie (supérieure à 1MeV). Au niveau du sol, on estime le flux de neutrons 1500 fois moins sévère qu'à une altitude de 12 km. Des dysfonctionnements de circuits intégrés dans les systèmes avioniques, à cause des neutrons, sont observés et enregistrés à ces altitudes depuis les années 1990 [Tab. 93]. Les pions et les muons ont une influence négligeable sur le fonctionnement d'un système vu leur faible interaction avec la matière et leur nombre insuffisant. Les protons d'énergies supérieures à quelques dizaines de MeV peuvent avoir les mêmes effets que les neutrons d'énergie équivalente.

Tableau I-2 : Comparaison de flux (cm-2. s-1) de particules [Gasio. 01].

Type de particule	I- Altitude (12000 m)	II- Niveau de la mer	Rapport (I/II)
Neutrons	9	$6\ 10^{-3}$	$1{,}5\ 10^{3}$
Protons	$2\ 10^{-1}$	$2\ 10^{-4}$	10^{3}
Electrons	3	$4\ 10^{-3}$	$7{,}5\ 10^{2}$
Muons	10^{-1}	$2\ 10^{-2}$	5
Pions chargés	$5\ 10^{-3}$	10^{-5}	$5\ 10^{2}$

I.3. Perturbations d'origine intentionnelle

Au-delà des effets environnementaux, des perturbations peuvent être générées volontairement et ont pour objectif d'inférer des informations sur les données sensibles présentes dans un circuit ; on parle alors d'attaques. En effet, la sécurité des systèmes intégrés est une préoccupation de plus en plus importante. Les systèmes intégrés sur puce se développent pour de très nombreuses applications de la vie courante, et manipulent de plus en plus de données sensibles voire confidentielles. Ces systèmes sont la cible d'attaques qui ont pour but de perturber le comportement du système afin d'obtenir des informations secrètes. Ce type d'attaques appelé attaques en faute (par exemple, DFA pour *Differential Fault Analysis*) a été introduit par Boneh et al. en 1997 [Boneh. 97] et consiste à provoquer une erreur de calcul. Par comparaison avec un calcul sans erreur, l'attaquant peut retrouver le secret (en général, une clé de chiffrement), ou du moins une partie du secret.

L'introduction de fautes dans un circuit peut être effectuée par la variation des conditions de fonctionnement standard ou par exposition à une source optique. Nous donnons dans cette section une vue globale sur les moyens d'injection de fautes connus dans la littérature.

I.3.1. Perturbation de la tension d'alimentation

Tout circuit est conçu de façon à tolérer une certaine variation d'alimentation électrique, typiquement de ±10% par rapport à la tension nominale. En dehors de cette marge de tolérance, le circuit ne fonctionne plus correctement. Les modifications de la tension d'alimentation ont des effets sur la vitesse de cheminement de données dans la logique combinatoire située entre des éléments de mémorisation [Zuss. 12]. En effet, lorsque les temps de propagation d'un bloc combinatoire deviennent supérieurs à la période d'horloge, la valeur capturée en sortie du bloc dans l'élément de mémorisation (registre) est erronée. Par conséquent, le circuit passe dans un état logique incorrect. Des pics de tension (couramment appelés *power glitches*) sont généralement utilisés afin de contrôler l'instant d'injection.

I.3.2. Perturbation de l'horloge

Les circuits synchrones cadencés par un signal d'horloge ont une fréquence de fonctionnement maximale. Celle-ci est définie en considérant la durée maximale d'un transfert de données entre deux éléments de mémorisation (chemin critique). Une violation des contraintes temporelles peut être obtenue si l'attaquant réussit à forcer une fréquence d'horloge supérieure à cette fréquence maximale (on parle alors d'*overclocking*). La période d'horloge devient alors plus courte que la durée de certains transferts dans le réseau combinatoire. Dans ce cas, des valeurs incorrectes sont échantillonnées par certains éléments de mémorisation. En augmentant globalement la fréquence de fonctionnement du circuit, le nombre d'erreurs augmente. Mais un grand nombre d'erreurs ne permet pas d'exploiter les résultats obtenus. Pour remédier à ce problème, ce procédé d'overclocking peut être appliqué de façon à induire des fautes sur un seul cycle d'horloge. On parle alors de perturbation transitoire du signal d'horloge (appelée aussi *clock glitches* s'il s'agit d'ajouter des transitions du signal) [Agoy. 10], ce qui permet de choisir le moment d'injection (cycle). Il est aussi possible de modifier de manière transitoire le facteur de forme de l'horloge. La figure I-3 présente un exemple de signal d'horloge modifié pouvant être utilisé pour une telle attaque.

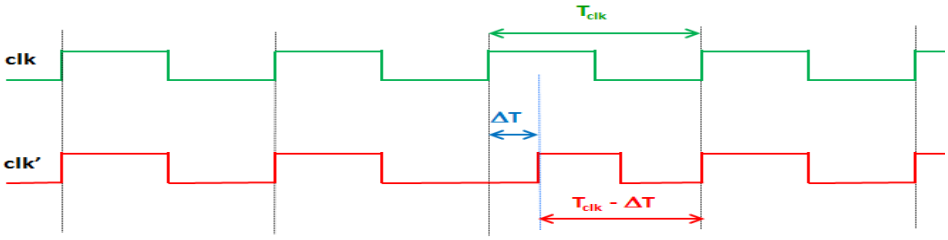


Figure I-3 : Modification locale d'une période d'horloge.

I.3.3. Perturbation de la température

Les circuits intégrés sont conçus pour fonctionner dans une certaine plage de température. La modification de la température opérationnelle d'un circuit, au-delà des bornes spécifiées par le constructeur, modifie son comportement. Les blocs de mémoire sont les éléments les plus sensibles à ce type de modifications. En effet, ces dernières peuvent altérer les temps de lecture et d'écriture dans la mémoire et donc être la source d'incohérences dans les données lors de l'exécution. L'étude faite dans [Skoro. 09] montre que des fautes ont été

injectées dans des mémoires Flash et EEPROM du microcontrôleur PIC16F628 par une augmentation de la température au moyen d'un spot lumineux.

I.3.4. Perturbations électromagnétiques

Au début des années 2000, les variations transitoires de l'environnement électromagnétique (*glitchs* ou impulsions électromagnétiques) ont été introduites par [Quis. 02]. Elles sont devenues par la suite un moyen d'injection de fautes de plus en plus répandu : [Schm. 07], [Dehb. 12], [Dehb. 13] et [Bay. 12] montrent qu'il est possible d'injecter des fautes en modifiant le champ électromagnétique autour d'un circuit. En effet, placer un circuit à proximité d'un fort champ électromagnétique crée un courant de Foucault qui modifie le nombre d'électrons à l'intérieur d'une grille d'oxyde de transistors [Ott. 04]. Cela modifie la tension de seuil du transistor de telle sorte qu'il ne peut plus commuter pendant la perturbation électromagnétique. Ainsi, suivant le type de transistor, ceci permet de s'assurer qu'une cellule mémoire contient la valeur 0 ou 1. Un matériel restreint suffit pour réaliser une attaque de ce type. Le courant de Foucault peut être créé en utilisant une bobine active avec suffisamment de puissance et la propagation du champ électromagnétique peut être faite à l'aide d'une antenne. Les dernières années ont montré différents effets possibles à partir de plusieurs types d'équipements.

I.3.5. Perturbations optiques ou par particules

Parallèlement aux radiations naturelles, qui sont issues d'évènements très énergétiques, plusieurs moyens ont été développés pour créer des particules pouvant affecter les circuits électroniques. Un rayonnement lumineux peut notamment être utilisé pour injecter des fautes dans un circuit. Un tel dispositif d'injection de fautes nécessite en général l'ouverture du boitier pour avoir une bonne pénétration des éléments perturbateurs. Nous présenterons ici le flash lumière, les accélérateurs de particules et les lasers.

I.3.5.1. Injection de lumière

La simple lumière visible peut faire réagir un composant électronique par effet photoélectrique. Un flash assez intense (quelques watts) peut être utilisé pour perturber un circuit électronique [And. 97]. Un équipement simple et peu coûteux a été présenté dans [Skoro. 03]. Celui-ci permet d'injecter des fautes en utilisant un simple flash d'appareil photo. Cela leur a permis de modifier la valeur d'un bit d'une cellule mémoire [Rand. 09]. Le principe est d'utiliser l'énergie du flash pour perturber le silicium du composant : cette énergie est en fait absorbée par les électrons du silicium, les autorisant ainsi à passer de la bande de valence à la bande de conduction, créant des paires électron-trou qui forment un courant photoélectrique local et peuvent rendre passant un transistor originellement bloqué par la valeur de sa grille. L'effet observable est une éventuelle modification du niveau de tension sur la sortie d'une porte logique ou l'inversion d'un point mémoire.

On ne peut toutefois irradier très précisément une faible surface car une lumière polychromatique a tendance à diffracter et est difficile à focaliser. De plus, on ne maîtrise absolument pas l'énergie rayonnée sur le silicium. Le flash lumière est facile à mettre en œuvre,

mais ne permet généralement pas une localisation précise de l'injection de faute et provoque donc un dysfonctionnement en de nombreux points du circuit attaqué.

I.3.5.2. Accélérateurs de particules

Les accélérateurs de particules peuvent aussi être utilisés pour bombarder des circuits avec un faisceau de particules énergétiques et injecter des fautes [Gunn. 89]. L'énergie et le type de particule sont contrôlables. Il s'agit habituellement de reproduire l'effet des particules présentes dans un milieu radiatif sur des composants fonctionnant dans cet environnement. L'injection de fautes est réalisée en utilisant des ions lourds ou des particules ionisantes (neutrons, particules alpha, etc.). Cette méthode est considérée comme le moyen de caractérisation de référence dans le milieu industriel pour déterminer la sensibilité des composants électroniques vis-à-vis des particules radiatives. Puisque le flux est très important, les données statistiques sur la fiabilité du composant peuvent être obtenues plus rapidement que dans un environnement réel. Ceci permet de simuler en accéléré le comportement d'un circuit en utilisation normale.

L'accès à des installations permettant ce type d'approche est limité. Chaque campagne d'injection nécessite une phase de préparation importante et des coûts substantiels. Cette approche est donc utilisée pour des caractérisations dans le domaine spatial, mais n'est pas réellement dangereuse dans l'optique de perturbations intentionnelles, d'autant plus qu'elle ne permet pas un bon contrôle spatial et temporel de l'injection de fautes.

I.3.5.3. Injection laser

Un laser (Light Amplification by Stimulated Emission of Radiation) est une source de lumière cohérente et monochromatique. Les lasers constituent une solution particulièrement bonne pour la photo-excitation d'un circuit électronique. Le petit rayon de leur faisceau, après focalisation par un microscope optique (possible grâce à sa cohérence), permet d'exciter le circuit sur une surface proche du micromètre carré. L'énergie du rayonnement lumineux utilisé est absorbée par le silicium du circuit. Lorsque l'énergie transmise est supérieure au seuil permettant à des électrons de passer dans la bande de conduction du silicium, des paires électron-trou sont alors créées le long du faisceau lumineux [Rosc. 13] [Duter. 11]. Ce phénomène est illustré sur la figure I-4. Ces paires électron-trou peuvent aboutir à l'apparition d'un courant photoélectrique au niveau d'un transistor. Ce courant entraîne alors l'apparition d'un pic de tension qui peut se propager dans des blocs de logique combinatoire. Le pic de tension induit peut entraîner l'apparition d'une faute s'il est échantillonné par un élément mémoire comme une bascule. Le pic de courant peut aussi faire basculer directement une cellule de mémorisation.

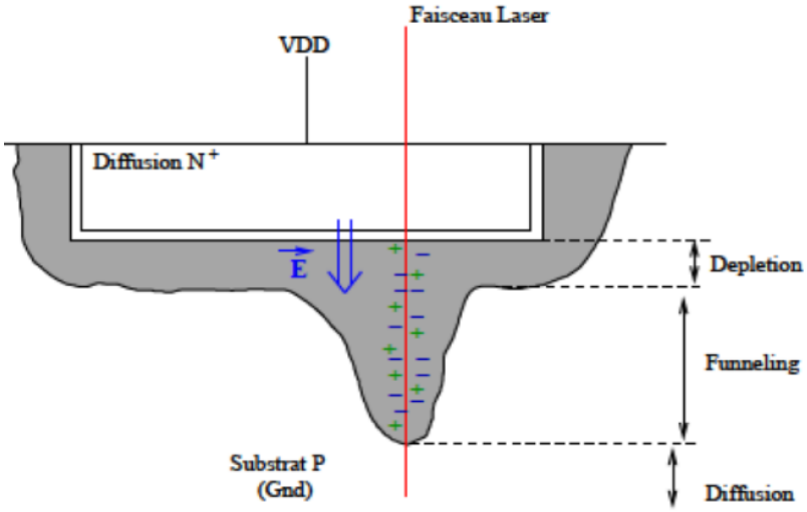


Figure I-4 : Apparition de paires électron-trou lors d'un tir laser sur une jonction PN [Rosc. 13].

La monochromaticité permet de bien contrôler la profondeur de pénétration des photons dans l'échantillon. Quand la longueur d'onde est suffisamment petite, l'essentiel de l'injection a lieu au voisinage de la surface. Au contraire, quand la longueur d'onde est suffisamment grande, l'absorption de photons s'effectue à plus grande distance dans l'échantillon. Les principaux avantages du laser sont qu'il permet une bonne maîtrise temporelle et spatiale de l'injection de fautes favorisant ainsi la reproductibilité de l'attaque, même si la réduction des motifs de gravure actuels complique le processus. L'autre avantage est que le coût des bancs laser est nettement inférieur à celui des accélérateurs de particules par exemple.

Dans [Rosc. 13], les auteurs ont utilisé une source laser pour injecter une faute dans une cellule SRAM. Par rapport à l'utilisation de lumière concentrée, le laser permet une bien plus grande précision lors d'une injection de faute [Caniv. 10]. Des effets sur certains blocs de logique d'un microcontrôleur ont également été obtenus dans [Trich. 10] ; ceux-ci ont permis de modifier l'exécution d'instructions assembleur sur le microcontrôleur ciblé.

I.3.6. Résumé des perturbations intentionnelles

Nous avons listé dans cette section différents moyens possibles pour un attaquant afin de perturber un circuit et injecter des fautes. Ces moyens d'attaque peuvent être classés selon leur degré de précision.

L'attaquant peut soit perturber le fonctionnement du circuit sans aucune précision (par ex., modification de la température, de la fréquence d'horloge ou de la tension d'alimentation) et on retrouve dans ce cas des fautes qui affectent un grand nombre d'éléments, soit perturber d'une façon plus précise dans le temps et dans l'espace (ex. laser) et la faute se trouve localisée dans ce cas sur l'élément ciblé ou quelques éléments proches.

Le risque avec la perturbation peu précise est que l'on peut créer tellement d'erreurs que le comportement fautif ne peut plus être exploité par l'attaquant. Mais pour qu'une perturbation très locale ait l'effet escompté, il faut une bonne connaissance de l'implantation physique du circuit attaqué.

I.4. Effets des perturbations sur les circuits intégrés

Les perturbations précédemment citées peuvent engendrer des effets temporaires ou des dommages permanents dans les matériaux qu'elles traversent. Ce paragraphe donnera un aperçu des mécanismes d'interaction entrant en jeu lorsqu'une particule ionisante traverse de la matière. On peut classifier deux effets distincts : la dose et les événements singuliers.

I.4.1. Effets de dose

Une partie importante des effets des radiations que l'on peut observer est regroupée dans ce qu'on appelle *effets de dose*. Ces effets résultent de l'interaction entre les particules et les isolants des circuits intégrés. En effet, les particules ionisantes sont susceptibles de déposer des charges dans les parties isolantes des composants électroniques. Ces charges s'accumulent dans le temps dans les oxydes. Il en résulte des effets tels que la diminution de la mobilité des porteurs et la création d'un champ électrique parasite. Cette accumulation de charges est définie par la dose qui est l'énergie par unité de masse déposée dans un matériau. Elle s'exprime en Gray. Un Gray est équivalent à l'absorption d'un joule par kilogramme (1 Gy = 1 J/kg). Les effets provoqués par la dose sur les composants électroniques sont nombreux et prennent différents aspects. L'un des principaux problèmes rencontrés concerne la dérive des paramètres électriques comme, par exemple, la modification de la tension de seuil des transistors de la technologie MOS (*Métal Oxyde Semiconducteur*), ce qui peut conduire à un état permanent passant ou bloquant [Sai. 98] [Ane. 00].

Ces effets peuvent être minimisés par blindage (protection par une feuille d'aluminium) et par l'utilisation de technologies appropriées (dites durcies). Dans le cas des équipements spatiaux, on fait en sorte que la durée de vie des composants soit supérieure à la durée de la mission. Les effets de dose ne sont alors pas critiques pour sa réussite.

I.4.2. Les effets singuliers (SEE)

Les évènements singuliers (*Single Event Effects*, SEEs) correspondent aux phénomènes déclenchés par l'interaction d'une particule chargée avec les matériaux semi-conducteurs. Le plus souvent, il s'agit de particules fortement énergétiques, telles que des ions lourds ou des protons énergétiques. En effet, une particule chargée peut déposer assez d'énergie et ioniser par la suite le milieu dans lequel elle passe en causant un effet sur le circuit. Les ions lourds causent une ionisation directe de la matière à travers laquelle ils passent laissant derrière eux des paires électron-trou qui, en se recombinant, donnent naissance à un SEE (figure I-5 (a)). Les protons quant à eux ne causent d'ionisation directe que dans les zones très sensibles du circuit. Cependant, généralement ils créent une réaction nucléaire qui peut produire un SEE par ionisation indirecte (figure I-5 (b)).

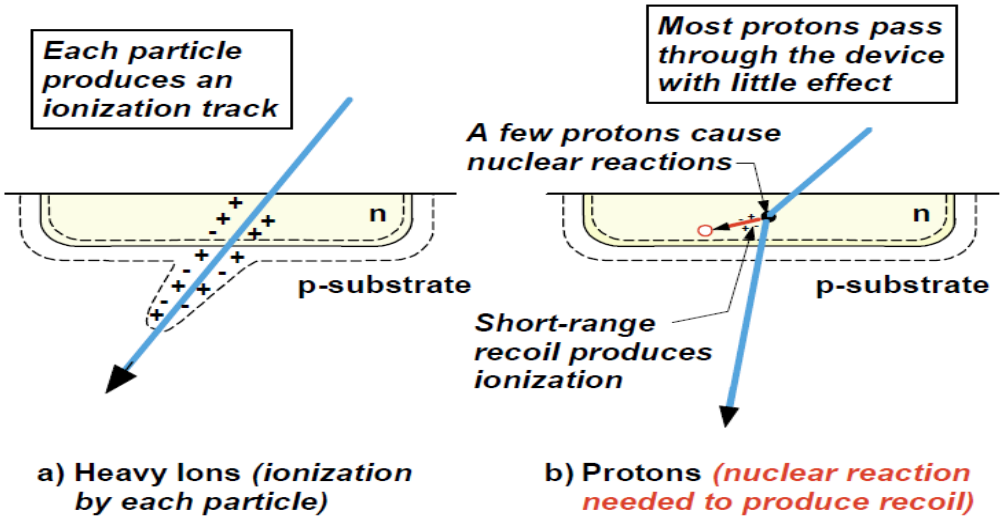


Figure I-5 : Ionisation du matériau [Kay].

Ces effets sont instantanés et peuvent provoquer des défaillances soit temporaires soit permanentes. C'est pourquoi ils sont généralement classés en deux sous catégories :

- Les évènements réversibles c'est-à-dire non destructifs appelés aussi fautes transitoires ou "*Soft Errors*". Ce type d'erreurs est lié à un phénomène transitoire dont les erreurs engendrées peuvent être corrigées.

- Les évènements irréversibles c'est-à-dire destructifs appelés aussi fautes permanentes ou "*Hard Errors*". Ces erreurs correspondent à la dérive importante d'un paramètre d'état (courant, tension, etc.) et peuvent conduire à la destruction d'un élément du circuit. Leurs effets sont alors permanents.

I.4.2.1. Événements singuliers irréversibles (hard errors)

- **SHE Single Hard Error (SHE)**

Un ion lourd par exemple (mais aussi un laser) peut déposer suffisamment d'énergie pour entrainer une grande variation de tension de seuil et rendre incontrôlable le transistor concerné. Ce phénomène intervient principalement dans les mémoires à haute intégration submicronique et a pour conséquence un bit collé.

- **Single Event Latch up (SEL)**

La proximité des transistors NMOS et PMOS dans les technologies CMOS induit la présence d'une structure parasite de type NPNP appelée thyristor entre l'alimentation et la masse. La mise en conduction de cette structure conduit a une amplification du courant avec un gain infini qui se traduit par un court-circuit entre les bornes d'alimentation. Le verrouillage de cette structure peut conduire à la destruction du circuit intégré si l'alimentation n'est pas immédiatement coupée [Fau. 05] [Buch. 97].

- **Single Event Gate Rupture (SEGR)**

Il s'agit de la rupture de grille d'un composant MOS après qu'il ait été irradié. Les SEGR sont déclenchés par des ions lourds lors de leur passage au travers de la couche isolante [Sext. 98]. Ces défauts ne sont pas électriquement actifs immédiatement après l'irradiation. Ils peuvent être activés après la fin de l'irradiation et sont alors identifiés comme des défauts précurseurs conduisant au déclenchement du claquage prématuré de l'oxyde de grille. Ils sont toujours destructifs pour les dispositifs.

- **Single Event Burnout (SEB)**

Ce phénomène prend place principalement dans les composants de puissance qui contiennent des centaines de transistors en parallèle. Un dommage d'un seul transistor peut conduire à la destruction de ses voisins par effet d'avalanche et rendre le circuit inutilisable.

- **Single Event Snap back (SES)**

Les SES sont rencontrés principalement dans les NMOS. Ils sont dus à la conduction d'un transistor bipolaire parasite entre le drain, le substrat et la source. Les technologies sous faible tension sont peu sensibles à cet effet [Norm. 94] [Jedec. 01].

I.4.2.2. Événements singuliers réversibles (soft errors)

- **Single Event Transient (SET)**

Cet effet concerne la logique combinatoire (ou les circuits analogiques). Un SET est initié lorsqu'une charge est déposée sur un nœud d'un transistor MOSFET, entrainant l'apparition d'un courant parasite qui se propage dans le circuit. Les conséquences de cet événement sont la génération de signaux transitoires indésirables qui peuvent perturber le fonctionnement des systèmes numériques.

Le pic de tension généré peut se propager dans le circuit jusqu'à être capturé, si certaines conditions sont réunies, par un ou plusieurs éléments de mémorisation tels que les bascules. Parmi ces conditions, on peut citer les temps de "setup" et "hold" de la bascule, l'amplitude de l'impulsion ainsi que la présence d'un front d'horloge. Si certaines conditions sont violées, la bascule peut entrer dans un état métastable. La fréquence d'horloge du circuit influence grandement la probabilité de capturer une impulsion transitoire.

- **Single Event Upset (SEU)**

Le terme de SEU est associé au phénomène entrainant le changement d'état (appelé *upse*t) d'un point mémoire en son état inverse sous l'effet d'une particule ionisante ou d'une attaque. Ce changement accidentel de niveau logique dans une mémoire est réversible (le point mémoire pourra être corrigé par le processus normal d'écriture) et ne conduit pas à la destruction du composant. Un SET peut se transformer en SEU s'il se propage à travers une chaîne logique jusqu'à un élément de mémorisation et qu'il est capturé sur le front d'horloge. Tout composant électronique possédant des points de mémorisation peut être

sensible au phénomène d'upset (mémoires DRAM ou SRAM, microprocesseurs, bascules, verrous, etc.)

- **Multiple Bit Upset (MBU) - Multiple Cell Upset (MCU)**

Un MBU survient quand une particule incidente ou une attaque crée plusieurs erreurs dans un même mot mémoire (SEU multiples mais localisés au niveau logique). Les charges produites par la particule se diffusent sur plusieurs jonctions voisines. La probabilité est plus faible que celle du SEU mais l'occurrence de ce phénomène devient de plus en plus fréquente avec la diminution de la géométrie des cellules mémoires. Il en résulte une complexification de la problématique de détection et de correction d'erreur dans les systèmes numériques. La propagation d'un événement transitoire (SET) peut aussi induire des erreurs multiples lorsqu'elle parvient à l'entrée de plusieurs points de mémorisation en satisfaisant leurs conditions d'écriture. Le MCU correspond de façon similaire à plusieurs SEUs, mais touchant des bits n'appartenant pas au même mot mémoire [Jedec. 07].

- **SEFI Single Event Functional Interrupt**

Le terme SEFI désigne les événements causés par une perturbation et menant à un blocage temporaire de fonctionnalité du circuit (ou à l'interruption de l'opération en cours). Les SEFI qui ne concernent que des événements n'endommageant pas le circuit peuvent durer jusqu'à la coupure d'alimentation dans certains cas ou un temps fini dans d'autres cas. Ils désignent en général un événement qui dure longtemps [Kog. 98] [Jes. 96]. Ces effets peuvent provenir d'un SEU survenant dans la partie de configuration d'un composant complexe. Le composant est alors inutilisable jusqu'à ce que sa mémoire soit réinitialisée.

I.5. Modèles de fautes ou d'erreurs

Dans le cadre de cette thèse, nous nous intéressons uniquement aux fautes transitoires et aux erreurs qu'elles entrainent (soft errors). Comme indiqué précédemment, une faute transitoire est déclenchée par des conditions environnementales telles que la fluctuation d'une ligne d'alimentation, une interférence électromagnétique, un impact de particule ou une attaque. Après une faute de ce type, le circuit peut continuer à fonctionner, aucun dommage permanent n'étant généré. Cependant, des résultats erronés peuvent avoir été générés pendant cet intervalle de temps, dont la gravité peut être très importante pour l'application. Afin d'étudier les conséquences de ces aléas, il est nécessaire d'élaborer des modèles de fautes ou d'erreurs permettant de représenter le plus fidèlement possible les phénomènes physiques relatifs aux effets transitoires.

D'un point de vue électronique, il existe de nombreux modèles de fautes représentant les fautes physiques à divers niveaux d'abstraction : électrique, transistor, porte logique, comportemental, fonctionnel, etc. Les modèles présentés dans cette section cherchent à représenter, au niveau logique (voire comportemental), l'impact naturel des particules énergétiques. Ceux-ci peuvent aussi représenter certains types d'attaques par injection de fautes. Par exemple, l'effet d'une attaque laser très focalisée sur un nœud interne du circuit est assez similaire à un SET ou un SEU. D'autres types d'attaques peuvent toutefois nécessiter

d'adapter le modèle de fautes, en particulier au niveau de la multiplicité spatiale des fautes engendrées, c'est-à-dire au niveau du nombre de bits pouvant être simultanément perturbés.

Un SET correspond à l'inversion temporaire d'un signal logique. Ce modèle correspond particulièrement bien aux phénomènes observés dans les environnements radiatifs et aussi aux effets du laser si le spot est centré dans une zone de logique combinatoire. La largeur d'impulsion obtenue dépend de la technologie, de la topologie des transistors, de la charge électrique collectée et de la capacité du nœud atteint ; elle est donc difficile à connaître précisément, ce qui implique la prise en compte d'un intervalle de temps, classiquement de quelques dizaines à quelques centaines de picosecondes dans un contexte d'injection de fautes naturelles, parfois plus pour certaines attaques. Pour un circuit synchrone, un SET est créé dans le réseau combinatoire et a la possibilité de se propager à travers lui jusqu'à un ou plusieurs éléments de mémorisation (figure I-6). En fonction des paramètres choisis, on pourra aussi ajouter un attribut de multiplicité temporelle aux fautes transitoires.

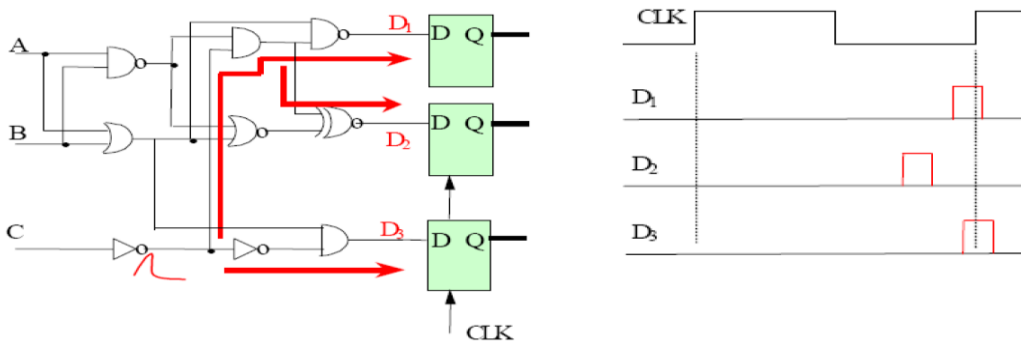


Figure I-6 : Scénario de propagation d'une impulsion transitoire dans un circuit combinatoire (a) et les formes d'ondes des entrées des bascules (b).

Un SEU correspond au basculement d'un point mémoire (appelé *bit-flip*), sans effet destructeur. Ce type de modèle peut s'étendre aux effets multiples (MBU et MCU) correspondant à la commutation simultanée de plusieurs points mémoires sous l'effet d'un impact unique. Dans le domaine de la sécurité, de telles fautes peuvent être obtenues suite à certains types d'attaques non focalisées, ou avec des technologies dont les dimensions sont très inférieures à la taille minimale de la zone perturbée.

Il existe d'autres types de modèles de fautes transitoires comme les fautes *Bit-set* et *Bit-reset*. Lorsqu'un bit est modifié et que sa valeur ne peut passer que de '0' vers '1' (respectivement de '1' vers '0') on parle alors de Bit-set (respectivement Bit-reset). Si l'erreur est créée, le même effet qu'un bit-flip est observé. Par contre, lorsque la valeur initiale du bit est déjà à '1' (respectivement '0'), aucune faute n'est injectée, ou tout du moins la valeur du bit reste la même. Ce comportement indique une forte dépendance aux données des fautes correspondant aux modèles Bit-set/Bit-reset.

Pour être applicables sur des descriptions comportementales, et donc assez tôt dans le flot de conception, les modèles de fautes ou d'erreurs précédemment cités se résument donc à des inversions de bits, les attributs suivants pouvant être envisagés :

- Durée : Elle est exprimée en nombre de cycles (pour les circuits synchrones) ou en durée effective en fonction du niveau d'abstraction choisi. La durée effective est dépendante de la nature de la faute. En général, les fautes naturelles sont exprimées en picosecondes ou sont limitées à un cycle d'horloge. La durée des fautes intentionnelles dépend du moyen d'injection employé, en général de quelques picosecondes jusqu'à des dizaines de nanosecondes, donc potentiellement un nombre assez élevé de cycles d'horloge.

- Multiplicité : Il est utile de définir pour chacun des modèles de fautes un degré de multiplicité spatiale et temporelle. La multiplicité temporelle correspond au nombre de répétitions ou à la fréquence de répétition d'une faute dans le circuit. La multiplicité spatiale correspond au nombre de points affectés simultanément par une injection de faute. En fonction du moyen d'injection de faute employé, la multiplicité peut être élevée, en particulier pour une injection par flash lumière ou par variation de l'horloge ou de l'alimentation, très peu précises. La répartition géographique des fautes sur le circuit dépend également du moyen d'injection : avec un laser par exemple, on peut considérer que les fautes sont localisées dans la même zone géographique, la zone sur laquelle est focalisé le laser. En revanche, une attaque par variation de l'horloge aura un effet beaucoup plus global sur le circuit ; la distribution géographique des fautes est alors très différente. On pourra considérer une distribution aléatoire ou une distribution déterministe non localisée sur une zone précise.

- Instant d'injection : le moment d'injection peut être défini selon plusieurs critères comme par exemple un temps absolu (pico ou nanosecondes …), ou en terme de cycles d'horloge, ou encore par rapport à un événement précis pendant le déroulement de l'expérience.

I.6. Robustesse et sûreté de fonctionnement d'un système

La sûreté de fonctionnement (*dependability*) d'un système est définie comme la propriété qui permet de placer une confiance justifiée dans le service qu'il délivre [Lapr. 04]. La définition est axée sur la confiance. En d'autres termes, la sûreté de fonctionnement d'un système est sa capacité à éviter les défaillances de service qui sont plus fréquentes et plus graves que ce qui peut être toléré par l'utilisateur. Elle s'appuie sur un ensemble de mesures qui, pendant toutes les étapes de la vie du produit, permettent de s'assurer que la fonctionnalité sera maintenue tout au long de la mission pour laquelle il a été conçu. La propriété de sûreté de fonctionnement est associée à plusieurs attributs, les principaux étant :

- Fiabilité (*reliability*) : la continuité de service est assurée pendant une durée minimale,

- Disponibilité (*availability*) : le pourcentage du temps pendant lequel le système est prêt à l'utilisation est supérieur à un certain seuil,

- Innocuité (*safety*) : la probabilité d'occurrence de défaillances jugées critiques est inférieure à un certain seuil,
- Sécurité (*security*) : la probabilité que la confidentialité et l'intégrité des données traitées soient maintenues.

Même si le terme "fonctionnement" est parfois omis, on parle alors simplement de "sûreté", il faut le garder en mémoire car il a une importance cruciale. En effet le niveau de sûreté d'un circuit dépend de l'environnement dans lequel il se trouve (exposition aux perturbations) mais également de la tâche qu'il doit effectuer et de sa charge de travail. Suivant sa charge de travail un circuit peut se comporter différemment en présence d'une ou plusieurs fautes. Le niveau de sûreté d'un circuit est évalué par rapport au service rendu. La notion de service rendu est donc déterminante. Si le service est tel que prédéfini en terme de fonctionnalité et de performance on dit qu'il est correct.

Une défaillance (*failure*) survient lorsque "le service délivré dévie du service correct, soit parce qu'il n'est plus conforme à la spécification, soit parce que la spécification ne décrit pas de manière adéquate la fonction du système" [Lapr. 04]. Une erreur, état anormal du système, peut entraîner une défaillance si elle se propage et devient observable de l'extérieur. L'origine d'une erreur est une faute dans le système, telle une valeur logique incorrecte dans un bloc combinatoire. Cet enchaînement est très souvent illustré par un schéma comme celui de la figure I-7. Si une faute survient dans une partie utilisée du circuit elle sera activée et engendrera, sous certaines conditions, une erreur c'est à dire une information (ou état) erronée mémorisée. Dans le cas contraire, si la faute se trouve dans une partie non utilisée du circuit à cet instant, elle est dite latente. Même en cas d'erreur, il n'y aura pas forcément de défaillance, soit parce que la déviation n'est pas significative du point de vue de l'application, soit parce que l'information erronée ne sera finalement pas utilisée.

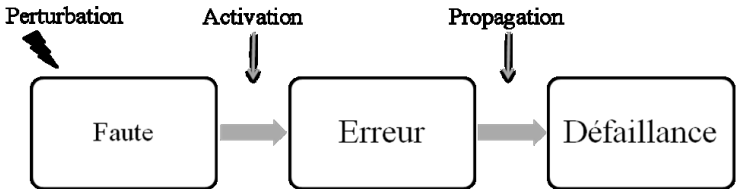


Figure I-7 : Schéma Faute-Erreur-Défaillance.

Une "soft-error" est un état erroné temporaire et n'engendre pas la destruction du circuit. En effet, ce dernier reprend son fonctionnement normal après la disparition de cette erreur. Par exemple, une telle erreur dans un registre disparaît après réécriture de ce registre. Cependant, avant son masquage, l'erreur peut s'être propagée en créant d'autres erreurs dans le système, pouvant conduire à une défaillance.

L'évolution des technologies et de nouvelles techniques d'attaques augmentant la probabilité de telles erreurs, comme expliqué précédemment, il s'avère nécessaire d'évaluer

précisément les conséquences potentielles au niveau fonctionnel sur l'exécution d'une application et d'identifier les éléments les plus critiques d'un système. Dans ce contexte, nous analyserons l'effet des inversions de bits, simples ou multiples, qui est reconnu comme étant le plus représentatif à la fois des évènements naturels et des attaques.

I.7. Conclusion

Dans ce chapitre, les différentes sources de perturbations menaçant la sûreté des circuits intégrés ont été décrites. Cette thèse s'intéresse aux fautes transitoires ayant lieu dans des systèmes intégrés numériques. Dans cette optique et à la vue des phénomènes divers à considérer nous avons présenté les modèles de fautes qui nous paraissent les plus probants dans le cadre de l'analyse de sûreté. Les fautes transitoires doivent désormais être prises en compte au niveau du sol et plus seulement pour les applications spatiales. Il est aussi important dans certains cas de prendre en compte des actions malveillantes. Malgré leurs modes d'action très différents, les diverses perturbations provoquent les mêmes effets au niveau comportemental. Par exemple, les particules chargées provenant de l'environnement radiatif naturel ou d'une source optique telle qu'un laser conduisent finalement à une inversion de bit (ou des inversions de bits) au niveau des éléments de mémorisation. Le même effet pourrait être observé par variation de la tension d'alimentation ou de la fréquence d'horloge du circuit, mais avec un effet plus global sur le circuit. Il y a donc des besoins forts pour analyser l'effet, au niveau applicatif, de ces erreurs potentielles. Des techniques d'évaluation de robustesse seront présentées dans ce document, pour répondre à ces besoins.

Avant cela, dans le chapitre suivant nous présentons l'état de l'art des techniques d'évaluation de robustesse. Cet état de l'art nous permettra de faire le point sur les avantages et les inconvénients des techniques connues et de proposer ensuite de nouvelles méthodes répondant le mieux possible aux problématiques actuelles dans le domaine de l'analyse de robustesse.

Chapitre II : Etat de l'art sur les techniques d'évaluation de robustesse

II.1. Introduction

La protection des circuits contre les effets des perturbations transitoires est désormais nécessaire pour assurer un comportement satisfaisant. En même temps, la protection de tous les éléments d'un circuit augmente le temps nécessaire à sa conception et à sa mise en place. De plus, elle est très coûteuse et peut mener dans certains cas complexes à des dégradations notables des propriétés (performances, surface, puissance consommée, etc.) ce qui est inacceptable pour des applications qui imposent de fortes contraintes. Par conséquent, afin de définir un meilleur compromis fiabilité/coût, les protections implantées doivent être optimisées de façon à réduire les coûts en matériel et en performances. Ainsi, le surcoût lié aux méthodes de protection des circuits doit être gardé dans des limites acceptables sur le plan économique. Pour cela différentes approches d'analyse de robustesse ont été proposées à différents niveaux du cycle de développement d'un système numérique, c'est-à-dire, depuis la phase de conception jusqu'à l'implémentation physique. Les techniques d'évaluation de robustesse permettent de caractériser la sensibilité globale des circuits et conduisent à identifier des zones critiques où l'apparition d'une erreur pourrait provoquer des défaillances dans le système. Ces zones doivent être alors protégées pour répondre aux exigences de l'application en termes de sûreté de fonctionnement ou de sécurité. L'objectif final est d'assurer une protection efficace et de réduire, simultanément, les coûts en priorisant les protections aux blocs les plus vulnérables.

Nous nous proposons ici de faire une présentation des techniques d'évaluation de robustesse pendant la phase de conception, c'est-à-dire, avant que le circuit soit fabriqué. Cette étude ne cherche pas à être exhaustive, mais à montrer les principales directions pouvant être suivies pour analyser la robustesse d'un circuit. Nous décrivons dans la section II.2 les différentes techniques d'injection de fautes. Puis, nous étudions dans la section II.3 quelques méthodes présentes dans la littérature portant sur l'analyse de robustesse des systèmes à base de microprocesseur.

Même si certaines approches présentées utilisent des implantations sur FPGA comme support d'analyse, notre travail n'a pas pour objectif d'évaluer les effets d'erreurs dans la configuration d'un FPGA SRAM. Les travaux ou méthodes proposés avec cet objectif ne seront donc pas cités. La présentation sera limitée aux approches visant à comprendre les effets d'erreurs survenant dans la logique du circuit que l'on cherche à implanter, notamment les bascules du circuit cible.

II.2. Techniques d'injection de fautes

Depuis la fin des années 80, l'injection de fautes s'est avérée être une technique efficace pour l'analyse de sûreté. Cette technique consiste en l'introduction intentionnelle de fautes dans

un système afin d'en mesurer la robustesse et/ou déterminer sa tolérance aux fautes. Le principe est de comparer le comportement nominal du circuit (sans injection de fautes) avec son comportement en présence de fautes, survenant lors de l'exécution d'une application.

Les techniques d'injection existantes peuvent être classifiées en deux catégories :

- Les approches d'injection qui s'appliquent au niveau du circuit physique lorsque celui-ci est déjà fabriqué. Nous citons par exemple : l'utilisation d'une source laser [Samps. 97] [Lewis. 05], la variation de l'alimentation [Karls. 91], la corruption de la mémoire [Miche. 94] ou encore l'injection de fautes au niveau broches d'E/S [Arlat. 94] [Madei. 94].

- Les approches qui permettent une analyse dans le flot de conception, typiquement par simulation ou par émulation matérielle (prototypage). Nous nous focalisons par la suite sur ce type d'approches.

II.2.1. Injection de fautes par simulation

La simulation permet l'analyse d'un système à plusieurs niveaux, depuis la description fonctionnelle jusqu'au comportement physique. Elle utilise des langages de description comme VHDL ou Verilog, parce qu'ils permettent de modéliser le circuit aussi bien au niveau comportemental (algorithmes) qu'au niveau structurel (portes logiques). À haut niveau d'abstraction (*High-level Description Language - HDL*), on peut vérifier le fonctionnement du circuit, avant qu'il soit synthétisé. La simulation logique niveau portes permet de vérifier les délais et les contraintes temporelles du circuit analysé. En contrepartie, une simulation niveau portes est au moins de 2 à 3 ordres de grandeur plus lente qu'une simulation comportementale. Des simulations sont possibles au niveau électrique (transistors) ou même physique (process) mais sont encore beaucoup plus lentes à réaliser et sont donc limitées classiquement à de petites fonctions, voire à un seul transistor.

L'injection de fautes en simulation consiste à simuler le comportement du circuit en présence d'un modèle de fautes choisi en fonction du niveau d'abstraction (inversion, collage, etc.). Dans cette partie, nous balayons les différentes méthodes d'injection de fautes par simulation. Nous nous concentrons surtout sur les approches d'injection de fautes par simulation comportementale (RTL ou *Register Transfer Level*) ou niveau portes car leur mise en œuvre est la plus réaliste pour nos objectifs.

II.2.1.1. Injection au niveau RTL

Au niveau transferts de registres (RTL), il existe deux façons d'injecter des fautes pendant la phase de simulation, soit en forçant la valeur de certains signaux internes à l'aide des commandes du simulateur, soit en utilisant une description du circuit que l'on a probablement modifiée afin d'injecter des fautes. Le processus de modification est appelé instrumentation et la description obtenue est dite description instrumentée.

- **Exploitation des commandes du simulateur**

Cette méthodologie nécessite un outil de simulation puissant, permettant d'arrêter la simulation et de forcer des nœuds internes dans le circuit. Lors de la simulation RTL du circuit à analyser, il est possible d'exploiter des commandes du simulateur ou des routines d'un langage spécifique (par exemple, TCL pour *Tool Command Language*). Un exemple d'outil est MEFISTO (pour *Multi level Error/Fault Injection Simulation Tool*) [Jenn. 94]. Ce dernier utilise la manipulation de signaux ainsi que celle des variables afin d'injecter des fautes. L'utilisation des commandes du simulateur permet de simuler le composant pas à pas et de modifier les signaux et les variables pendant la simulation, pour un temps défini.

Un autre outil de d'injection par simulation RTL a été développé au sein de l'université de Turin [Corn. 00]. Cet outil tire profit des mécanismes de debug des simulateurs commerciaux pour injecter des fautes et exploite l'interface proposée par le langage TCL. En fonction du modèle de fautes choisi, l'analyse de la description comportementale (VHDL RTL) produit une liste de fautes. Le "simulateur de fautes" est composé d'un ensemble de routines qui interagissent avec le simulateur en utilisant la liste de fautes précédemment générée. Une étude a été faite dans [Berro. 02] afin d'étendre l'approche pour inclure les fautes de type bit-flips. Dans l'optique d'accélérer les campagnes d'injection, le nombre de fautes a été réduit grâce un prétraitement appliqué sur les cibles d'injection.

L'agence spatiale européenne a de son côté conçu un outil similaire permettant l'injection de fautes au niveau RTL en utilisant la commande "force" du logiciel de simulation ModelSim, ainsi que les langages TCL et Perl [Guti. 04].

Les outils d'injection par simulation au niveau RTL fonctionnent de la manière suivante : la simulation se déroule normalement jusqu'à un point d'arrêt appelé instant d'injection. A cet instant, les processus sont arrêtés et la (ou les) fautes sont injectées en modifiant la représentation de l'état interne du circuit (valeur des bascules, valeur des signaux, etc.). La simulation continue et l'évolution de l'état du circuit après l'injection de la faute et le résultat final sont analysés pour diagnostiquer le comportement du circuit.

L'injection de fautes en utilisant les commandes du simulateur a l'avantage de ne pas modifier le circuit.

- **Simulation avec description instrumentée**

La simulation d'une description instrumentée permet de simuler des fautes à différents endroits dans le circuit. L'instrumentation se traduit par l'ajout de signaux de contrôle et d'éléments spécifiques permettant l'injection de fautes. Deux approches différentes peuvent être adoptées pour modifier la description initiale du circuit : les saboteurs et les mutants.

Les saboteurs sont des blocs supplémentaires qui, lorsqu'ils sont activés, peuvent altérer la valeur ou les caractéristiques temporelles d'un ou plusieurs signaux [Jenn. 94] [Franc. 06]. Il s'agit d'une modification structurelle de la description originale du circuit. L'insertion des saboteurs est faite entre les blocs/composants existants du circuit afin d'assurer l'injection de

fautes pendant la simulation. L'utilisation des saboteurs peut servir pour injecter certains types de fautes dans les interconnexions modifiées. Cependant, il est presque impossible d'injecter des fautes 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.

MEFISTO-L [Boue. 98], développé au sein du laboratoire LAAS, est un outil d'instrumentation à base de saboteurs. Il utilise ainsi des sondes pour observer l'état des signaux. La description originale est analysée afin de générer les modèles de fautes, les cibles potentielles et l'ensemble des signaux à observer. L'approche se base sur un simulateur commercial pour la compilation et la simulation de la description.

Un mutant correspond à une version modifiée d'un ou de plusieurs blocs/composants originaux du circuit [Leve. 00]. Cette fois-ci, il s'agit d'une modification comportementale de la description originale. Quand l'injection est inactive, le mutant se comporte exactement comme le bloc remplacé. Dans le cas contraire, il imite un comportement fautif. Généralement, les modèles de fautes utilisés par des mutants sont plus complexes que ceux utilisés par les saboteurs. Ils peuvent être statiques si leur configuration reste la même durant toute la simulation. Ils peuvent aussi être dynamiques ; en fonction de la valeur d'un signal le bloc mutant sera activé ou non. De cette manière, il est possible d'injecter des fautes permanentes et transitoires. Il faut noter que cette méthodologie augmente le temps de simulation. L'utilisation de mutants ralentit la simulation de manière plus significative que l'utilisation de saboteurs.

Une approche pour la génération de mutants a été proposée dans [Hadj. 05]. Dans ce travail, l'instrumentation est effectuée en modifiant finement la description RTL des blocs du circuit à analyser, contrairement à l'approche à base de saboteurs qui rajoute des blocs entre les blocs existants. L'approche permet d'injecter des fautes de type SEU, MBU ou MCU dans n'importe quel bloc, ce qui n'est pas possible avec des saboteurs, mais la modification de la description RTL du circuit est beaucoup plus difficile à automatiser.

II.2.1.2. Injection au niveau logique

Parallèlement aux approches au niveau comportemental, des outils ont été développés pour injecter des fautes au niveau portes logiques (après synthèse). Parmi ceux-ci nous pouvons citer FAST, VERIFY, VFIT, ROBAN et LIFTING.

Le premier outil qui est apparu dans la littérature, **FAST** pour *FAult Simulator for Transients* [Cha. 96], est un environnement d'injection de fautes transitoires au niveau portes. Il se compose en fait de deux simulateurs distincts. Le premier simule le circuit jusqu'à l'instant où la faute injectée est enregistrée par une cellule mémoire. Les caractéristiques des portes logiques utilisées (temps de propagation, temps de montée, etc.) ont donc une importance non négligeable lors de cette première phase. A partir du moment où une erreur est mémorisée, le second simulateur observe les effets de cette erreur sur les sorties du circuit. L'utilisation de deux simulateurs a pour but d'accélérer la simulation après que la faute ait été mémorisée puisqu'à partir de cet instant il n'est plus nécessaire d'avoir une grande précision temporelle. Au niveau portes, afin d'obtenir un modèle de fautes qui soit le plus proche d'un SEU, des

simulations électriques (SPICE) ont été menées. Ces simulations ont permis d'avoir des informations précises comme les délais de portes, la durée nécessaire de la faute et plus généralement le comportement des bascules (*Flip- Flops*) et des verrous (*latches*).

Le second outil, **VERIFY** pour *VHDL-based Evaluation of Reliability by Injecting Faults efficientlY* [Volk. 97] permet l'injection de fautes au niveau portes logiques et RTL. Un type de signal interne est ajouté aux types déjà existants en VHDL ; il s'agit d'une extension du langage. Ce type contient le temps moyen entre chaque faute et la durée moyenne de chaque faute. Chaque faute est associée à un composant, ceci est donc transparent par rapport aux autres composants. L'interface du simulateur permet d'activer les fautes dans les différents composants. VERIFY permet l'injection d'inversions de bits et de collages à 1 ou 0. L'outil nécessite un compilateur et un simulateur dédié pour comprendre et utiliser les nouveaux types permettant de décrire les fautes. Les deux ont été développés au sein de l'outil VERIFY. Le compilateur extrait les signaux d'injection et les lie à l'exécutable pour la simulation. L'approche nécessite cependant une modification du code qui peut être très contraignante.

Ensuite, **VFIT** [Baraz. 00] est un outil d'injection de fautes fonctionnant sous Windows. Il a été réalisé autour du simulateur commercial ModelSim. Cet outil est capable d'injecter des fautes dans des modèles VHDL au niveau portes logiques, registre et circuit. Les modèles de fautes pouvant être injectés sont des collages et des inversions de bits, ces dernières pouvant être permanentes, transitoires ou intermittentes. L'outil peut utiliser trois techniques d'injection : par commande du simulateur, saboteur ou mutants. Une fois l'injection et la simulation réalisées, VFIT peut effectuer deux types d'analyses. La première analyse classifie les fautes et les erreurs ainsi que leurs incidences et délais de propagation. La deuxième analyse vérifie si le système est tolérant aux fautes ; la détection et les mécanismes de récupération sont alors validés. Cet outil est actuellement en cours d'être retravaillé. En effet, pour le moment VFIT ne fonctionne que sous Windows XP avec une ancienne version de ModelSim. L'outil ne gère pas sa mémoire de manière optimale, il est donc impossible de réaliser une injection sur un modèle complexe.

L'outil suivant, **ROBAN**, développé par la société iRoC Technologies, est un outil qui permet l'injection et la simulation rapide de fautes transitoires. Il s'intéresse principalement aux fautes ayant lieu dans la logique combinatoire d'un circuit [Angh. 00] [Alex. 02]. Il a été intégré avec des simulateurs commercialisés tels que ModelSim et NCSim, ce qui offre à la fois un simulateur logique et un injecteur de fautes via une seule interface. Un environnement d'injection de fautes basé sur ROBAN et sur un modèle SET a été présenté dans [Alex. 04]. Son objectif est d'évaluer la probabilité d'occurrence de "soft-errors" dans les différents registres du circuit. Il permet ainsi de connaitre l'évolution du SER (pour *Soft Error Rate*) en fonction de quelques paramètres comme, par exemple, la fréquence d'horloge. Afin d'estimer la propagation et de simuler l'impact d'une faute, le simulateur utilise la liste d'interconnexions existantes dans le circuit. Les contraintes temporelles relatives aux portes logiques sont données dans un fichier SDF (pour *Standard Delay Format*) donnant les informations sur les délais les plus élevés, les plus faibles et les délais nominaux. Chaque scénario sera utilisé lors de la simulation. Le défi majeur est la difficulté d'obtenir une probabilité réaliste. Pour cela, un grand

nombre de vecteurs de test en entrée est nécessaire, ce qui implique un temps de simulation élevé. La solution principale consiste à injecter plusieurs fautes pour le même vecteur de test en découpant les différentes parties de la logique combinatoire du circuit. Le temps de simulation est réduit de manière conséquente ce qui permet à l'outil d'être performant. Il est donc possible de simuler un grand nombre de fautes avec beaucoup de vecteurs de test, ce qui permet d'obtenir une probabilité plus précise et réaliste.

Le dernier outil, **LIFTING** [Bosio. 08], est un outil disponible gratuitement permettant de réaliser des simulations logiques simples, ainsi que des simulations de fautes. Il permet aussi d'analyser les résultats de simulation. L'outil prend en entrée la description en Verilog de la liste d'interconnexions d'un circuit numérique ainsi que la liste des vecteurs à injecter en entrée. Il est possible de spécifier la liste de fautes à injecter, dans le cas échéant l'outil considérera tous les lieux d'injections. Les types de fautes disponibles sont les collages simples/multiples et SEU (bit flip dans un élément de stockage. Ce simulateur possède la particularité d'utiliser une approche orientée objet. L'outil est réalisé de manière très flexible afin de pouvoir être modifié aisément pour convenir à diverses utilisations. Les rapports d'injection générés par LIFTING sont détaillés. Pour une séquence de test donnée, l'outil va générer la séquence de sortie de référence ainsi que chaque séquence de sortie pour chaque faute injectée [WW. 3].

II.2.2. Injection de fautes par émulation

La simulation permet de réaliser des analyses avec beaucoup de flexibilité. D'un autre côté, elle nécessite des temps expérimentaux très élevés ou des outils de calcul puissants, surtout dans le cas des circuits complexes qui embarquent des applications ayant besoin d'un grand nombre de cycles pour s'exécuter. Pour cela, des techniques basées sur l'émulation matérielle ont été proposées, tout d'abord sur la base d'émulateurs commerciaux [Leve. 99], puis en utilisant surtout des plateformes configurables de type FPGA. Les émulateurs commerciaux ont à nouveau été utilisés plus récemment [Dave. 09].

D'un point de vue analyse de robustesse par injection de fautes, l'émulation représente un gain de temps significatif par comparaison à la simulation (de plusieurs ordres de grandeur, selon le type de campagne d'injection souhaité). Autre avantage : la reconfiguration dynamique est un atout qui peut être exploité pour l'injection de fautes. L'utilisation du prototypage s'est largement répandue avec les améliorations des circuits programmables récents (vitesse de fonctionnement, ressources disponibles, etc.). Généralement l'environnement d'injection de fautes par émulation se compose d'un ordinateur hôte et d'une plateforme configurable (carte FPGA). L'ordinateur hôte exécute des modules logiciels permettant à l'utilisateur d'interagir avec le FPGA par échange de données via des périphériques.

Deux approches se distinguent pour l'injection de fautes dans un environnement FPGA : l'instrumentation du circuit prototype et la reconfiguration.

II.2.2.1. Injection par instrumentation

Comme pour la simulation, pour injecter des fautes par instrumentation, il est nécessaire de modifier le circuit original, le synthétiser et ensuite émuler la version modifiée.

L'environnement d'injection de fautes présenté dans [Vanh. 08] met en œuvre un prototype matériel d'une version instrumentée du circuit à analyser. Il comprend trois niveaux d'exécution dont un niveau logiciel embarqué qui permet d'accélérer les expériences en conservant une grande flexibilité : l'utilisateur peut obtenir le meilleur compromis entre complexité de l'analyse et durée des expériences. [Vanh. 08] propose également de nouvelles techniques d'instrumentation et de contrôle des injections afin d'améliorer les performances de l'environnement. Une évaluation prédictive de ces performances renseigne l'utilisateur sur les paramètres les plus influents et sur la durée de l'analyse pour un circuit et une implantation de l'environnement donnés.

De nombreux autres travaux ont été réalisés à travers le monde sur des principes similaires ; il serait trop long de tous les détailler. Nous citons par ailleurs l'outil FIFA (pour *Fault Injection by means of FPGA*) [Cive. 01a] développé au sein de l'institut Polytechnique de Turin. C'est un outil d'injection de fautes à base de FPGA qui effectue des injections de type *bit-flips* dans les cellules mémoires d'une description instrumentée du circuit à analyser. Il a été ensuite amélioré afin de mettre en œuvre des techniques dédiées aux circuits de type microprocesseur [Cive. 01b] [Cive. 03]. L'outil est composé de trois modules. Les deux premiers sont purement logiciels et exécutés exclusivement sur l'ordinateur hôte. Ils ont pour objectif de générer la liste des fautes à partir de la *netlist* du circuit, les vecteurs d'entrée et d'analyser les résultats. Le troisième est un module mixte logiciel/matériel. Celui-ci contient l'outil d'instrumentation, le contrôleur des injections et l'interface matérielle émulée sur le FPGA qui applique les commandes du contrôleur. Le taux d'erreurs estimé à l'aide de cette méthode est assez précis car l'instrumentation permet d'accéder à tous les éléments du processeur, ce qui se traduit par une bonne contrôlabilité spatiale. Cette méthode est cependant intrusive et elle nécessite d'avoir suffisamment de ressource dans le FPGA pour ajouter l'instrumentation.

II.2.2.2. Injection par reconfiguration

Pour les FPGAs basés sur les cellules SRAM, le fichier de configuration contient les données ainsi que les commandes de configuration. Les cellules SRAM de configuration peuvent être vues, en simplifiant, comme un registre à décalage. Les données de configuration sont téléchargées sur le FPGA de façon sérielle depuis l'ordinateur hôte. Pour certains composants, ces données peuvent ensuite être relues à partir du FPGA ainsi que le contenu de toutes les cellules mémoire ; ces deux opérations sont appelées respectivement *readback* et *capture*.

Une fois qu'un FPGA a été configuré, il existe deux manières de le reconfigurer : de façon statique (Compile-Time Reconfiguration - CTR) ou de façon dynamique (Run-Time Reconfiguration - RTR). Dans le cas d'une reconfiguration statique (CTR), pour implémenter une quelconque modification il est nécessaire de recompiler et re-synthétiser les blocs modifiés, de régénérer le fichier de configuration pour le système complet et de télécharger cette nouvelle configuration sur le FPGA. La reconfiguration dynamique (RTR) offre la possibilité de reconfigurer certains FPGA durant le déroulement d'une application. Il est possible de reconfigurer tout le circuit (Global RTR) ou parfois uniquement une partie spécifique de celui-ci, on parle alors de reconfiguration partielle (Local RTR). Lors d'une reconfiguration partielle,

le sous-ensemble non reconfiguré peut, avec certains FPGA, continuer à fonctionner normalement.

L'injection de fautes par reconfiguration se fait par modification directe des données de configuration (*bitstream*), sans nécessiter de modifications de la description du circuit, donc sans instrumentation. L'un des avantages est d'éviter le risque d'une erreur lors de l'instrumentation, conduisant à des erreurs lors de l'évaluation de sûreté. Le deuxième avantage est le gain de ressources, permettant d'évaluer des circuits plus complexes sur un FPGA donné.

Le principal objectif de la reconfiguration dynamique est d'économiser le temps induit par les différentes étapes de la reconfiguration statique (compilation, synthèse, génération du fichier de configuration complet). De plus, contrairement au cas de la reconfiguration statique, l'injection de fautes peut être déclenchée à tout moment pendant l'exécution d'une application sur le FPGA. Les données de configuration peuvent être conservées lors de la première configuration ou bien relues sur le FPGA (*readback*). Cette technique permet d'injecter des collages à 1 ou 0 permanents ou transitoires dans les blocs combinatoires, ceci en modifiant par exemple les tables de configuration (*Look-Up Table* - LUT). Il est possible aussi d'injecter des SEUs dans les cellules mémoires ; ceci requiert la lecture préalable de leur contenu (*capture*). Les mécanismes de reset (Global Set/Reset) de ces points mémoires sont alors utilisés pour en inverser la valeur.

Les expériences présentées dans [Anton. 00] ont été effectuées avec deux FPGA Xilinx distincts, l'un d'entre eux disposant de mécanismes de reconfiguration partielle. Les bitstreams (fichiers de configuration) sont modifiés avec JBits, un outil Java qui permet une lecture de la configuration du FPGA et une reconfiguration simple et rapide. Les résultats obtenus démontrent la faisabilité de l'approche et montrent que celle-ci permet des économies de temps importantes dans certaines conditions. Il semble cependant que de nombreuses contraintes ont été rencontrées. Une description plus complète de l'approche ainsi que des résultats obtenus sont proposés dans [Anton. 03].

Les travaux réalisés dans [Kafk. 06] et [Sterp. 07] ont cherché à accélérer les expériences d'injection en réduisant les temps de reconfiguration induits par la reconfiguration partielle. Ils supposent que des circuits sur le même FPGA puissent être utilisés pour contrôler la configuration des autres parties du FPGA. En effet, ils proposent d'utiliser un processeur embarqué inclus dans le FPGA pour effectuer les reconfigurations. Ce processeur exploite les capacités internes de configuration avec le port ICAP (pour *Internal Configuration Access Port*) disponible dans les FPGA de Xilinx. Cette technique est appelée endo-reconfiguration et permet de réduire l'échange de données avec le PC hôte, conduisant ainsi à de meilleures performances lors de la campagne d'injection de fautes. Depuis, de nombreux travaux ont utilisé cette approche. Les résultats présentés comme référence dans ce document ont été obtenus avec une plateforme utilisant ce principe et nommée ATE-FIT5 [WW. 4].

[Leve. 10] propose aussi une nouvelle méthodologie d'injection de fautes, basée sur l'abstraction de motifs d'erreurs, obtenus à partir d'expériences de pré-caractérisation physiques

notamment pour des plateformes reconfigurables FPGA. La méthode a été implémentée dans un environnement d'injection de fautes utilisant l'endo-reconfiguration.

D'autres travaux de recherche ont porté sur l'utilisation de la reconfiguration dynamique partielle. Nous citons par exemple l'outil FT-UNSHADES [Agui. 05] développé par l'université de Séville. Il s'agit d'un environnement d'injection de fautes de type bit-flips dans des FPGA Xilinx de la famille Virtex. Cet environnement met en œuvre deux versions du circuit à analyser ; une de référence et une pour l'injection. A la fin de l'exécution, l'analyse de l'effet des fautes injectées se fait par comparaison des états des deux versions. L'utilisation de deux copies du circuit accélère la comparaison des résultats obtenus avec les résultats de référence mais réduit énormément l'espace disponible sur le FPGA.

II.2.3. Comparaison des deux approches

L'injection de fautes par simulation permet de faire l'analyse de sûreté d'un circuit à différents niveaux d'abstraction au cours de son développement. Les deux faits importants à noter sont que plus on descend dans les niveaux d'abstraction plus les campagnes d'injection seront longues mais plus les modèles de fautes seront précis.

Pendant la simulation RTL, l'injection de fautes dans un circuit peut s'effectuer suivant deux techniques : l'utilisation des commandes de simulateur ou l'instrumentation. La première technique met en œuvre les capacités de *debug* et de précision incluses dans les simulateurs. Conséquemment, elle est dépendante du simulateur utilisé ainsi que des langages qu'il supporte. La deuxième technique est indépendante du simulateur puisqu'elle se base uniquement sur la modification de la description originale avant simulation du circuit analysé. Par contre, apporter des modifications judicieuses sur une description matérielle nécessite un travail supplémentaire. Cette tâche est difficile à automatiser car elle dépend du langage de description et de la façon dont le circuit est décrit. La simulation de fautes au niveau portes logiques offre des modèles de fautes plus précis puisque la structure exacte du circuit est connue. Le principal inconvénient de cette approche est qu'elle est très gourmande en temps.

Le prototypage pour l'analyse de sûreté s'est imposé avec l'évolution des performances des circuits programmables (fréquence de fonctionnement, ressources disponibles, etc.). Ces derniers intègrent aussi des composants et des fonctionnalités qui peuvent être utilisés pour accélérer l'injection de fautes (ex. reconfiguration dynamique partielle, cœurs de processeurs, interfaces de configuration, etc.). De ce fait, l'émulation matérielle permet de réaliser des expériences d'injection de fautes plus rapides que la simulation.

La technique d'instrumentation permet de rester indépendant du type de circuit programmable, mais nécessite une version modifiée de la description initiale du circuit à analyser. La version instrumentée doit néanmoins rester synthétisable, ce qui augmente les temps de préparation des campagnes d'injection. L'utilisation des mécanismes de reconfiguration offre un gain en temps puisqu'elle évite tout changement dans le code du circuit. Cependant, la mise en œuvre et les protocoles liés à ces mécanismes (lecture/chargement de la configuration) varient en fonction du FPGA choisi, ce qui limite la portabilité des approches développées. Le gain en temps dépend aussi, d'après [Leve. 03], des

caractéristiques de la plateforme configurable (débit de reconfiguration, contrôle des signaux de set et reset, etc.) et de la connectique avec l'ordinateur hôte. De plus, si le circuit à prototyper est assez complexe (cas d'un véritable SoC constitué par de nombreux blocs matériels), la quantité de logique programmable et le nombre d'entrées/sorties disponibles sur le FPGA peuvent être des facteurs limitants surtout s'il s'agit d'une approche basée sur l'émulation de deux copies du système (avec faute et référence).

Le tableau II-1 présente un résumé des avantages et des inconvénients qui caractérisent les deux types de techniques d'injection présentés dans cette section.

Tableau II-1 : Récapitulatif des avantages et des inconvénients des approches d'injection.

Technique	Avantages	Inconvénients
Simulation	+ Modélisation multi-niveau : du système au transistor. + Contrôlabilité et observabilité totales. + Faible coût d'implantation : aucun matériel particulier n'est exigé.	+ Temps de simulation très important. + Outils de simulation puissants requis. + Nécessite la modification du code en cas d'instrumentation.
Emulation	+ Nettement plus rapide que la simulation. + Exploitation des fonctionnalités des FPGAs (reconfiguration dynamique, processeurs embarqués, etc.). + Possibilité de connecter le prototype dans son environnement opérationnel. + Possibilité d'implémenter les vecteurs d'entrée sur le FPGA pour encore augmenter la vitesse.	+ Nécessite un code synthétisable. + Nécessite la modification du code en cas d'instrumentation. + Coûts d'un FPGA et des logiciels de contrôle + Nombre de ressources configurables et d'entrées/sorties limité par le FPGA. + Mise en œuvre coûteuse en temps pour des systèmes complexes (préparation)

Peu importe la technique d'injection utilisée, il est fondamental de prendre en compte le temps relativement important pour lancer des campagnes exhaustives d'injection de fautes. Ce type d'expériences assure une analyse de robustesse très précise, mais il est devenu rapidement impossible en pratique sur les circuits actuels. Prenons l'exemple d'un système avec B bascules et une application exécutée en N cycles. Si le modèle de fautes utilisé est le SEU, il faudrait alors effectuer (B * N) injections et donc (B * N) simulations (en plus de la simulation de référence). Ceci est devenu impossible en pratique sur les circuits actuels, sauf à disposer d'une ferme de calcul particulièrement performante. Le temps nécessaire pour faire ces expériences est prohibitif même en utilisant l'émulation. Pour contourner ce problème, il est nécessaire d'utiliser la méthode d'injection de fautes statistique (ou SFI pour *Statistical Fault Injection*). Le terme statistique signifie que seulement un sous-ensemble des fautes possibles est injecté. Ce sous-ensemble est choisi de manière aléatoire par rapport aux cibles disponibles dans le modèle RTL (registres, verrous, etc.) et aux cycles d'injection. Ce processus permet au concepteur de fixer le nombre d'injections en fonction du temps disponible pour l'évaluation de robustesse. Le travail effectué dans [Leve. 09] propose des injections de fautes statistiques avec une quantification de l'erreur induite sur le résultat. Le nombre d'injections à réaliser peut ainsi être

considérablement réduit tout en maîtrisant la marge d'erreur en fonction du niveau de confiance souhaité.

II.3. Evaluation de robustesse des systèmes à base de microprocesseur

Dans le cas d'un système construit autour d'un microprocesseur exécutant du logiciel applicatif, l'analyse de robustesse peut être traitée par plusieurs approches spécifiques (en plus des méthodes d'injection de fautes présentées précédemment) et avec des points de vue différents. Le but de ces approches est toujours d'identifier les éléments les plus critiques. Nous détaillons dans la suite les principales méthodes évoquées dans la littérature.

II.3.1. Analyse par mesure de l'AVF (Architectural Vulnerability Factor)

II.3.1.1. Estimation du taux d'erreurs

La robustesse des systèmes électroniques est usuellement quantifiée par des métriques comme le MTTF (*Mean Time To Failure*) ou le MTBF (*Mean Time Between Failure*). Dans le cas des fautes transitoires, il est plus souvent fait référence au FIT (*Failure-In-Time*), qui correspond au nombre de défaillances moyen en 10^9 heures d'utilisation (soit 114.155 années). Il est souvent supposé que toute erreur conduit à une défaillance, donc le FIT se déduit directement de la complexité du circuit et des expériences permettant, pour une technologie donnée, de quantifier le taux de *soft errors* (SER - *Soft Error Rate*) [Hazu. 00]. Pour un système (intégré ou non) le FIT total correspond alors à la somme des FIT des composants ou blocs assemblés.

En 2003, [Mukher. 03] propose une approche permettant d'affiner la quantification du taux d'erreurs vu par l'utilisateur du système. Cette approche part du fait qu'une faute transitoire survenue dans le système peut être masquée au niveau architectural et, par conséquent, ne pas avoir d'effet au niveau applicatif. Les auteurs ont introduit la notion de facteur de vulnérabilité de l'architecture ou encore *Architectural Vulnerability Factor* (AVF), comme la probabilité qu'une faute dans une structure/composant du microprocesseur se traduise par une erreur visible dans le résultat final du programme exécuté (i.e. erreur d'application). Le taux d'erreurs effectif (FIT effectif) d'une structure est obtenu en faisant le produit de son AVF et de son taux d'erreur brut (FIT brut). Ceci permet d'obtenir dans certains cas des prédictions nettement moins pessimistes du taux d'erreur applicatif. Par exemple, cette méthode s'adapte particulièrement bien pour un microprocesseur avec un fonctionnement spéculatif. Une erreur lors d'une prédiction de branchement sera naturellement corrigée par les mécanismes existant. La logique du bloc de prédiction n'est donc pas critique du point de vue fonctionnel et son AVF peut être considéré nul, même si une *soft error* dans ce bloc peut entraîner une dégradation de performances.

II.3.1.2. Méthode d'analyse ACE

Le calcul de l'AVF a fait l'objet de différents travaux de recherche dans le monde. Nous nous concentrons dans cette partie sur la méthode analytique basée sur le concept d'ACE (*Architecturally Correct Execution*). Cette approche a été introduite initialement dans [Mukher. 03]. Les bits ACE sont les bits permettant une exécution correcte. Quand ces bits sont corrompus à un moment donné de l'exécution, ils provoquent une erreur d'application.

L'intégrité de ces bits est donc nécessaire pour une exécution correcte sans erreurs. Par opposition, les bits un-ACE (*unnecessary for Architecturally Correct Execution*) sont ceux dont la corruption n'influe pas sur le résultat final du programme exécuté. L'AVF d'un bit contenu dans une structure de microprocesseur est alors la fraction de temps pendant laquelle ce bit est qualifié comme ACE bit (formule (1)).

$$AVF\ of\ bit = \frac{ACE\ Time\ of\ Bit\ (in\ Cycles) \times 100}{Total\ Time\ (in\ Cycles)} \quad \textbf{(1)}$$

L'AVF d'une structure matérielle contenant n bits est obtenu en faisant la moyenne des AVFs relatifs aux différents bits. Ceci présente, en d'autres termes, la fraction de temps pendant laquelle la structure contient des bits ACE (formule (2)).

$$AVF\ of\ the\ structure = \frac{[\sum_{For\ all\ Bits} ACE\ Time\ of\ Bit\ (in\ Cycles))] \times 100}{Total\ Time\ (in\ Cycles) \times Total\ Number\ of\ Bits} \quad \textbf{(2)}$$

Théoriquement, calculer l'AVF revient donc à faire une analyse des durées de vie des points mémoires de la structure cible. Une telle analyse s'appuie sur l'identification de la nature de chaque bit (ACE/un-ACE). Toutefois, ce concept a été utilisé pour qualifier le niveau de sensibilité d'un processeur, indépendamment de l'application exécutée. Il s'agit donc d'une évaluation générique, qui peut être vue comme une "moyenne" ou une "probabilité" que le bit erroné ait un impact sur le résultat de l'exécution.

En pratique, l'estimation de l'AVF se fait soit en suivant la trace de chaque bit ACE tout le long du code à l'aide d'un modèle de performance, soit en analysant la pertinence de chaque structure. Cette technique est fondée sur la connaissance de la microarchitecture et l'identification des bits qui n'induiront pas d'erreurs s'ils sont modifiés. Tous les bits sont supposés être ACE jusqu'à preuve du contraire. Cette classification se fait en utilisant des règles d'architecture et de microarchitecture. A titre d'exemple, l'AVF d'un bloc de prédiction de branchement est nul car tous les bits contenus dans cette unité sont considérés comme un-ACE durant l'exécution du programme. En effet, une mauvaise prédiction aura pour effet de charger des instructions inutiles, qui devront être annulées lorsque le résultat du branchement est connu. En revanche, un SEU dans le compteur programme provoque une mauvaise instruction à exécuter, ce qui va affecter certainement le résultat du programme. Dans ce cas, l'AVF de cette structure vaut 100% puisque tous les bits peuvent être a priori ACE à tous les instants de l'exécution. Des exemples plus complexes (unités d'exécution, file d'attente des instructions, etc.) ont été traités pour des processeurs mettant en œuvre l'architecture IA-64 (*Intel Architecture 64 bits*) [Mukher. 03] [Weav. 04].

En outre, de nombreuses études ont été menées sur le facteur de vulnérabilité des blocs mémoires (cache de données, buffer de translation de données, banc de registres, etc.). La plupart repose sur l'estimation des durées de vie des variables critiques et nécessite un modèle détaillé du processeur. La notion d'analyse de durées de vie pour des cellules mémoires a été introduite par [Bisw. 05] afin d'identifier les mots ACE du cache. Comme illustré à la figure II-2,

le principe général consiste à diviser le cycle de vie d'un mot mémoire en fonction de ses différents états (lecture, écriture) pour repérer les intervalles de temps un-ACE où une corruption des bits ne sera pas critique. Dans ce modèle, tous les intervalles de temps qui conduisent à un accès en écriture sont un-ACE puisque l'accès en écriture écrase une erreur transitoire. Inversement, une erreur dans un intervalle de temps menant à un accès en lecture est toujours propagée à la sortie de la mémoire et risque de détériorer le résultat final du programme. Par conséquent, ces intervalles sont supposés être des intervalles ACE.

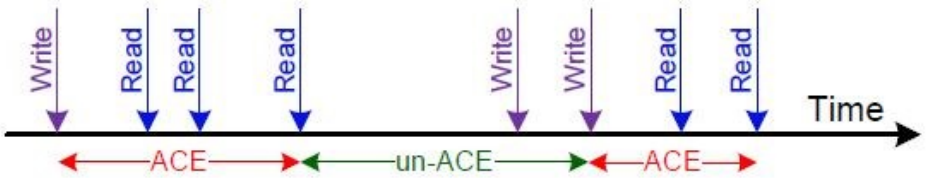


Figure II-1 : Intervalles ACE et intervalles un-ACE.

La méthode d'analyse ACE est capable d'estimer rapidement le facteur de vulnérabilité de plusieurs structures de microprocesseur. En effet, le modèle de performance permet d'effectuer plusieurs simulations dans un laps de temps réduit. Cette technique peut également être utilisée très tôt dans le flot de conception même avant la disponibilité des modèles RTL détaillés des structures. Toutefois, seulement les structures décrites dans le modèle de performance sont prises en compte. L'absence de détails d'implémentation peut conduire à un manque de précision qui se traduit par une surestimation de la criticité (ou une sous-estimation dans certains cas, ce qui peut être plus grave). Les expériences d'injection menées dans [Geor. 10] montrent que l'analyse ACE surestime parfois la vulnérabilité de certaines structures jusqu'à un facteur 10 en comparaison avec la méthode SFI (section II.2.3). Cette dernière s'est révélée être plus précise mais beaucoup moins rapide. Il est aussi important de souligner que la plupart des modèles de performance étant des logiciels propriétaires et spécifiques à des processeurs (généralement des processeurs Intel), il est difficile pour un utilisateur final de mettre en œuvre l'analyse ACE.

Bien que la méthodologie AVF puisse être efficace pour analyser la robustesse des microprocesseurs, elle est difficilement transportable à tout circuit numérique où aucun mécanisme d'exécution spéculative n'existe par exemple. De plus, cette approche ne tient pas compte des caractéristiques réelles de l'application exécutée car aucune distinction n'est faite entre le masquage intrinsèque des fautes au niveau du matériel et le masquage qui est introduit par l'application.

II.3.2. Analyse de robustesse par analyse du code

Des études se sont aussi intéressées à l'analyse de robustesse des applications indépendamment des détails de la microarchitecture qui représente le support de l'exécution. Ces études se basent sur l'analyse du code de la partie logicielle pour identifier les variables les plus critiques. Pendant l'exécution, ces variables se situent soit en mémoire, soit dans des registres du processeur. Traditionnellement, vu leur grande taille, seules les mémoires comme la mémoire principale et les caches ont été considérées comme critiques pour la protection

contre les erreurs logicielles. Cependant, [Blome. 06] a observé que pour un processeur intégré, la majorité des erreurs qui affectent l'état du processeur proviennent du banc de registres. En effet, comme ces registres sont lus très fréquemment, les données corrompues peuvent se propager rapidement à d'autres parties du système, ce qui augmente les chances de produire un état erroné. Les mémoires sont systématiquement protégées par des codes de parité ou des codes correcteurs d'erreurs. La protection de la totalité des registres en utilisant de telles stratégies réduit fortement les performances du système [Slege. 99], en particulier, en termes de puissance dynamique consommée. C'est pourquoi l'analyse de vulnérabilité du banc de registres est désormais nécessaire pour assurer une protection partielle des registres selon leur valeur de criticité [Blome. 06] [Montes. 07].

La criticité d'un registre donné dépend tout d'abord des variables qui lui sont assignées durant l'exécution. Cela dépend des caractéristiques de l'application, mais également du processus de compilation qui définit l'utilisation des ressources dans l'architecture cible. Une variable critique est une variable qui, en cas de modification, peut causer un blocage d'application ou un comportement inattendu du système. La criticité d'un registre dépend alors du pourcentage du temps pendant lequel celui-ci contient des variables critiques. L'identification de ces variables passe par la définition et le calcul de critères de criticité. Ces derniers diffèrent d'une approche à une autre mais certains sont assez récurrents comme la durée de vie ("lifetime") et le nombre de descendants ("fanout"). La manière dont ils sont calculés n'est toutefois pas toujours la même. Nous allons donc résumer ici les principales méthodes proposées pour estimer la criticité par analyse de code.

II.3.2.1. Analyse statique du code

L'analyse statique ne nécessite pas la simulation ou l'utilisation d'un système réel. Elle est basée sur des paramètres qui sont déterminés après un traitement effectué sur le code source ou pendant la phase de compilation du logiciel.

Dans [Bens. 00], les auteurs définissent une nouvelle métrique afin de caractériser la criticité des variables d'une application développée en C/C++. Cette métrique est appelée Poids-fiabilité (*Reliability weight*) et est calculée pour chaque variable manipulée par le programme. Deux paramètres sont utilisés pour le calcul : la durée de vie de la variable et ses dépendances fonctionnelles. La durée de vie (Dv) d'une variable est mesurée en comptant le nombre de lignes de code qui séparent une instruction d'écriture de la dernière instruction de lecture de la même variable ou de la fin de l'exécution du programme. La récursivité et les itérations ne sont donc pas prises en compte par cette méthode. Cette dernière ne tient non plus compte du nombre de cycles nécessaires à l'exécution d'une instruction donnée qui est lié à l'architecture cible et à ses caractéristiques (gestion des dépendances de données, pénalité des cache miss, etc.). En ce qui concerne les dépendances fonctionnelles, elles sont évaluées à partir du graphe de dépendance des variables (*Variable Dependency Graph* – VDG) dont chaque nœud représente une variable et chaque liaison signifie la dépendance entre les deux variables. Par exemple, la liaison vi → vj existe si et seulement si vj est une descendante de vi. Ce graphe est utilisé pour calculer un facteur D_f (v). Il suffit ensuite d'appliquer la formule (3) pour déterminer le paramètre Poids-fiabilité(v).

$$\text{Poids - Fiabilité}\ (v) = Kl * Dv(v) + Kw * \Sigma\, \text{Poids - Fiabilité}\ (\text{descendants}(v)) \quad (3)$$

où "Kl" et "Kw" sont des coefficients de normalisation.

[Lee. 09] introduit le concept de RFV (*Register File Vulnerablity*). Il s'agit d'une analyse statique pour estimer la vulnérabilité du banc de registres vis-à-vis des erreurs logicielles. La vulnérabilité se base sur un calcul complexe de durées de vie qui dépend des chemins suivis pendant l'exécution. Cette dépendance est analysée au niveau des blocs, ce qui permet de décomposer la vulnérabilité d'un bloc en deux types de vulnérabilités. La vulnérabilité intrinsèque (v_i) est la moyenne des longueurs des intervalles [écriture-lecture] dans le bloc et la vulnérabilité conditionnelle (v_c) représente la longueur moyenne des derniers intervalles [écriture/lecture – fin]. Combinant ces deux paramètres avec les probabilités post-condition S, le taux RFV est obtenu par la formule (4).

$$\sum_j f_j V_j = \sum_j f_j (v_{ij} + v_{cj} S_j) \quad (4)$$

où f_j et V_j sont respectivement la fréquence d'exécution et la vulnérabilité du $j^{ième}$ bloc.

[Berg. 10] propose une méthode permettant de réaliser une analyse rapide de criticité pendant la compilation. Ceci peut garantir une indépendance par rapport au langage de programmation de l'application. Une fois la compilation terminée, le compilateur fournit le fichier exécutable pour le processeur cible et des informations sur la criticité des variables. Pour chaque variable, la criticité est calculée en fonction de trois critères : sa durée de vie, ses dépendances fonctionnelles et le poids dans les conditions de branchement. Le premier critère de criticité est la durée de vie des variables. La méthodologie de calcul de ce critère est une méthode statique qui consiste à profiter des analyses précises existant dans les compilateurs modernes. Ensuite, le second critère de criticité est relié aux dépendances fonctionnelles entre les variables. La technique utilisée permet de calculer d'une manière optimisée ce critère. Elle définit l'ensemble des descendants directs d'une variable "v", autrement dit toutes les variables dont la valeur utilise la valeur de v. Partant d'un tel ensemble, il est possible de déduire l'ensemble de tous les descendants de v. Puis, le dernier critère se rapporte au poids dans les conditions de branchement. C'est le degré de participation d'une variable donnée dans les différentes conditions de branchement. Ceci concerne principalement les variables utilisées dans les instructions de branchement conditionnel. En effet, même si sa durée de vie est faible, une variable peut influer directement sur le flot de contrôle et donc sur le résultat final du calcul (par exemple, un compteur dans une boucle détermine le nombre d'itérations). En conséquence, plus ce critère est élevé, plus la variable est critique. Le calcul de ce critère se fait à l'aide du graphe GFC (Graphe de Flot de Contrôle) représentant l'organigramme du programme d'application. Enfin, la criticité d'une variable est obtenue en appliquant la formule (5).

$$\begin{aligned} \text{Criticité}\ (v) = {} & Kl * Cl(v) + Kw * Cw(v) \\ & + Kd * \sum_{z \subset desc(v)} M(v, z) * (Kl * Cl(z) + Kw * Cw(z)) \end{aligned} \quad (5)$$

où M est une matrice qui définit la dépendance de la variable v par rapport à tous ses descendants, C_l et C_w sont respectivement la durée de vie et le poids dans les conditions de branchement de la variable v, et Kl, Kd et Kw sont des coefficients de régulation respectivement associés à la durée de vie, à la dépendance fonctionnelle et au poids dans les conditions de branchement. Ils peuvent être utilisés pour mettre l'accent davantage sur un critère plutôt que les autres en fonction du type d'application.

II.3.2.2. Analyse dynamique du code

Par opposition à l'analyse statique, l'analyse dynamique du code se fonde sur l'utilisation des informations recueillies au cours de la simulation ou le fonctionnement du système pour estimer les facteurs de criticité. Elle peut aussi être faite à partir d'outils de "profiling".

L'étude faite dans [Patt. 05] se base sur la construction d'un graphe de dépendance (Direct Dependency Graph - DDG) à partir du code assembleur et des scenarii d'exécution. Le but est de représenter le plus fidèlement possible les dépendances entre les variables. En fait, le DDG capture les dépendances dynamiques entre les différentes valeurs produites pendant l'exécution d'un programme. En cours d'exécution, chaque affectation d'une variable est traitée comme étant une nouvelle valeur. Une valeur est donc toute variable ou toute adresse mémoire utilisée dans le programme pendant son exécution. Si une variable est réécrite, elle est considérée comme une nouvelle valeur. Cette technique résout le problème d'interdépendance des variables relevé dans les VDG mais implique un grand surcoût mémoire. En s'appuyant sur le DDG, le calcul de criticité est effectué en utilisant deux critères, à savoir la durée de vie et le nombre de descendants. Mis à part ces critères classiques, les auteurs ont défini de nouveaux critères tels que l'exécution, la propagation et la couverture.

- La durée de vie est la distance maximale entre un nœud et son successeur immédiat en termes d'instructions dynamiques,
- Le fanout d'un nœud est l'ensemble de tous les successeurs immédiats de ce nœud dans le DDG,
- L'exécution est la fréquence d'exécution des instructions statiques associées à des nœuds,
- Le taux de propagation d'une erreur pour un nœud donné est le nombre de nœuds qui peuvent être affectés par cette erreur avant de causer un crash,
- La couverture est le nombre de nœuds à partir desquels une erreur peut se propager vers un nœud donné avant de causer un crash.

La principale limitation de cette approche est l'absence de méthodes automatisées pour calculer les métriques. De plus, aucune formule regroupant ces différentes métriques n'a été proposée, il est nécessaire d'exécuter l'application pour générer le DDG et la dépendance potentielle de celui-ci vis-à-vis des valeurs d'entrée utilisées lors de cette exécution n'est pas analysée.

[Srihd. 08] a introduit la notion de PVF (*Program Vulnerability Factor*) afin de quantifier la vulnérabilité des applications mais sans tenir compte de l'architecture matérielle à l'opposé de

l'AVF (section II.3.1) qui tient compte de l'architecture sans tenir compte du programme. Le PVF peut être utilisé pour expliquer le comportement d'un programme en présence d'erreurs transitoires indépendamment du processeur utilisé. Ainsi, il offre aux programmeurs la possibilité de mesurer la vulnérabilité des différents segments de leurs programmes. Le calcul du PVF se base sur l'analyse du flux d'instructions pendant l'exécution. Cela signifie que toutes les propriétés dynamiques du programme sont prises en compte (nombre d'itérations, récursivité, nombre d'opérations de lectures/écritures effectuées sur les registres, etc.). Pour un programme ayant I instructions, le PVF d'une ressource architecturale "R" (banc de registres par exemple) contenant n bits est défini par la formule (6).

$$PVF_R = \frac{\sum_{i=0}^{I} (ACE\ bits\ in\ R\ at\ instruction\ i)}{B_R \times |I|} \quad (6)$$

Les bits ACE de "R" sont les bits dont la corruption pourrait potentiellement affecter le résultat de l'exécution de l'instruction "i".

[Restr. 14] exploite les mêmes métriques de criticité que celles utilisées dans [Berg. 10]. D'ailleurs, la formule (5) a été employée pour calculer la criticité des registres. Néanmoins, cette fois-ci, les métriques ne sont pas calculées par le compilateur mais à travers l'analyse dynamique du code pendant la simulation du système. Traditionnellement, la durée de vie d'un registre donné correspond à la somme des intervalles [écriture – dernière lecture]. Dans [Restr. 14], les auteurs affinent cette définition en proposant le terme "durée de vie effective". Ce critère représente la somme des intervalles [écriture effective – dernière lecture] et prend en compte le fait qu'une opération d'écriture dans un registre ne s'effectue pas instantanément. Ceci dépend fortement de l'architecture cible et des scénarii d'exécution. Par exemple, en cas d'absence d'aléas dans une architecture pipeline de 5 étages, l'écriture de la donnée est réalisée effectivement après 5 cycles d'horloge. En ce qui concerne les dépendances fonctionnelles, la matrice de dépendances est mise à jour à chaque instruction simulée. Le nombre de descendants directs d'un registre change également tout au long de la simulation. De même, le poids dans les conditions de branchement varie en fonction du type et de l'état du branchement.

II.4. Conclusion

Dans ce chapitre, nous avons cherché à présenter les grandes lignes des différentes techniques d'analyse de robustesse d'un système numérique. Principalement, nous avons survolé trois approches d'évaluation différentes. La première est une approche basée sur l'injection de fautes. Cette technique a l'avantage d'être applicable sur tout circuit numérique. De plus, les méthodes de mise en œuvre de cette technique 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. Cependant, l'analyse de ces méthodes a identifié des inconvénients dont le principal est le temps requis par les campagnes d'injection.

Les microprocesseurs sont un cas particulier de système numérique, pour lequel des méthodes spécifiques ont été proposé. En particulier, une approche repose sur l'affinement du

taux d'erreurs dans une architecture par l'estimation de son AVF. Plusieurs travaux de recherche ont proposé des outils et des modèles pour calculer l'AVF. Cette approche permet d'évaluer la criticité des différents blocs d'un composant mais est essentiellement efficace dans le cas des microprocesseurs dits "haute performance". De plus, elle se base uniquement sur la microarchitecture et ne prend pas en compte les caractéristiques de l'application exécutée. Elle est donc très utile pour affiner la susceptibilité générale d'un composant par rapport à son SER (*"Soft Error Rate"*) "brut", c'est-à-dire lié uniquement à la technologie et à l'environnement, mais ne permet pas de tenir compte des ressources effectivement utilisées par une application donnée. La dernière approche décrite dans ce chapitre porte sur l'évaluation de robustesse par analyse du code. Contrairement à l'approche AVF, cette dernière s'intéresse seulement à la partie logicielle. Les techniques qui dérivent de cette approche sont capables de fournir des informations sur la criticité des registres architecturaux pendant la phase de compilation ou par analyse dynamique. Mais la confiance qui peut être accordée à ces résultats est limitée car la criticité dépend fortement des spécificités de la microarchitecture. D'autres approches, intermédiaires, ont été proposées dans le cas des microprocesseurs, fondées par exemple sur une analyse de probabilité de bonne exécution, après une pré-caractérisation de l'architecture par injection de fautes [Savi. 12]. Dans tous les cas où l'injection de fautes n'est pas utilisée, la microarchitecture n'est cependant pas vraiment prise en compte.

Pour pallier les différentes faiblesses des techniques discutées dans ce chapitre, nous allons proposer deux nouvelles approches dans les chapitres suivants. La première approche est destinée aux systèmes à base de microprocesseur. La seconde approche sera plus générale, en permettant une évaluation de robustesse sur tout circuit numérique synchrone. Dans les deux cas, l'objectif est de pouvoir analyser précisément les ressources critiques pour une application donnée, avec une précision comparable aux injections de fautes statistiques mais avec une durée d'analyse suffisamment courte pour permettre des itérations pendant la conception. Un autre objectif est de pouvoir réaliser ces analyses dans le flot classique de développement, sans nécessiter de compétences ou de matériel spécifique comme par exemple dans le cas des injections de fautes par émulation.

Chapitre III : Approche d'évaluation de robustesse des microprocesseurs fondée sur l'analyse de criticité des registres

III.1. Introduction

Un nombre croissant d'applications repose aujourd'hui sur des circuits intégrés complexes et, plus largement, sur l'utilisation de systèmes embarqués incluant du matériel (un ou plusieurs cœurs de processeurs) et du logiciel (système d'exploitation et logiciel d'application). La confiance pouvant être accordée à de tels systèmes dépend de leur capacité à réagir de façon sûre (c'est à dire, non dangereuse pour l'application) lorsque des fautes surviennent pendant l'exécution. Dans ces systèmes, l'analyse de robustesse revient à évaluer précisément la criticité des différents registres par rapport aux données qu'ils contiennent pendant l'exécution de l'application. Ceci est compliqué dans le cas des processeurs récents à cause de l'évolution des architectures des processeurs et l'implémentation de mécanismes d'amélioration des performances.

Dans la section suivante, nous proposons une approche permettant d'évaluer la robustesse de microprocesseurs, lors de l'exécution d'une application donnée. Comme cas d'étude, notre choix s'est porté sur un processeur récent et assez complexe : le LEON3. Ce processeur a un pipeline optimisé, et est disponible sous la forme d'une description synthétisable, ce qui nous permet de réaliser des injections de fautes pour quantifier l'efficacité de notre approche. La section III.2 résume les principes de l'approche proposée. La section III.3 décrit les caractéristiques de l'architecture du processeur cible. L'approche proposée est mise en œuvre dans la section III.4 et un algorithme de prédiction de criticité des registres est présenté. Pour la validation, la section III.5 présente une étude comparative entre les résultats obtenus par l'algorithme développé et ceux issus de plusieurs campagnes d'injection de fautes. Les sections III.6 et III.7 seront consacrées à l'étude de l'impact des options de compilation et de plusieurs caractéristiques architecturales sur la criticité globale d'une application. Ces évaluations mettront par ailleurs l'accent sur les avantages apportés par la nouvelle approche.

III.2. Présentation générale de l'approche

La robustesse d'un circuit à base de microprocesseur est, comme précédemment mentionné, fortement liée à la criticité des registres utilisés pendant l'exécution d'une application donnée. La criticité d'un registre se définit par la probabilité qu'une faute survenant sur ce registre ait des effets sur le bon déroulement du programme exécuté, ou autrement dit, le pourcentage de cas dans lesquels une faute dans ce registre induit une erreur en fin d'exécution du programme. Par exemple, si un registre a une criticité de 50%, cela signifie que si une faute transitoire survient dans ce registre, il y a une chance sur deux que le résultat final du

programme soit erroné. En conséquence, dans le cas d'une protection sélective, ce sont les registres ayant une valeur de criticité élevée qui doivent être protégés en priorité.

L'aspect clé de l'évaluation de criticité des registres est l'analyse de leur durée de vie. La durée de vie d'un registre correspond au nombre de cycles pendant lesquels une faute survenant dans ce dernier conduit à une erreur d'application. De ce fait, plus la durée de vie d'un registre est grande, plus il est critique.

La criticité des registres varie en fonction de leur utilisation par l'application considérée. Elle dépend aussi de la microarchitecture puisque celle-ci définit les ressources matérielles qui seront utilisées selon les instructions de l'architecture cible (ISA-*Instruction Set Architecture*). Les évaluations classiques ne tiennent compte que des registres visibles par l'utilisateur. Ce sont les registres de l'architecture situés généralement dans le banc de registres et utilisés par le jeu d'instructions au niveau assembleur. Toutefois, dans les processeurs récents, de nombreux registres internes existent dans les unités d'exécution (unité entière, unité flottante, coprocesseur, etc.). Ceux-ci ont aussi un impact sur la robustesse du système.

Afin de mesurer précisément la criticité du système, il est nécessaire de prendre en compte à la fois les caractéristiques de l'application (partie logicielle) et les spécificités de la microarchitecture du processeur qui représente le support de l'exécution (partie matérielle). Dans cette optique, l'approche que nous proposons consiste à faire une analyse dynamique du comportement de tous les registres lors de l'exécution d'une application. Cette analyse a pour but de décrire les transferts d'information dans l'ensemble des registres en tenant compte des propriétés de l'architecture interne.

L'approche est basée sur (1) la réutilisation des informations provenant de la validation fonctionnelle d'une application donnée et (2) la modélisation du comportement du processeur pour chaque instruction en tenant compte des spécifications de la microarchitecture. La première étape consiste à obtenir une trace détaillée de l'état du processeur lors d'une simulation fonctionnelle. Cette trace peut être générée suite à une modification de la description RTL si l'on utilise une approche instrumentée, ou par l'intermédiaire de scripts de simulation afin d'accéder à l'état des registres internes. Le but ici n'est pas de modifier les valeurs de certains signaux, mais d'enregistrer les valeurs contenues par quelques registres qui définissent l'état du processeur à chaque cycle d'horloge. Une fois que l'état du processeur est obtenu pour une version donnée du logiciel d'application, un programme d'analyse est lancé afin de calculer la durée de vie, et donc la criticité, des registres généraux (dans le banc de registres) et internes (situés dans la microarchitecture). Ce programme modélise essentiellement l'utilisation des registres par chaque instruction. Il détermine ainsi les cycles auxquels un registre contient une information critique pour l'exécution de l'application. L'analyse faite peut alors être exploitée pour identifier les registres les plus critiques (en moyennant les cycles "critiques" de chaque registre) ou évaluer globalement la sensibilité de l'application vis-à-vis des perturbations (en moyennant les cycles "critiques" de tous les registres). La figure III-1 illustre le principe général de l'approche proposée.

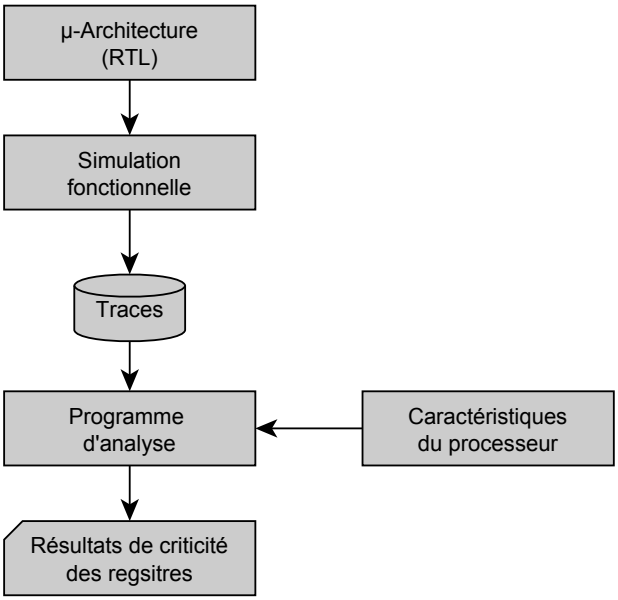


Figure III-1 : Vue globale de l'approche d'analyse de criticité proposée.

III.3. Le véhicule d'étude : le processeur LEON3

III.3.1. L'architecture SPARC v8

Dans cette partie, nous décrirons les principales caractéristiques de la spécification SPARC v8. L'annexe A.1 donnera plus de détails sur cette architecture.

L'architecture SPARC (*Scalable Processor ARChitecture*), dans sa version 8, est une architecture de processeurs RISC (*Reduced Instruction Set Computer*) 32 bits, conçue et produite par Sun Microsystems à partir de 1991. SPARC est une norme : IEEE-1754. Toutes les instructions et tous les registres ont une longueur de 32 bits. Il est néanmoins possible d'accéder en mémoire à des données dont la taille est de 8, 16, 32 ou 64 bits, ce dernier type d'accès utilisant deux registres consécutifs.

SPARC est une architecture de type pipeline avec des branchements et des appels de procédures retardés. Une instruction qui suit immédiatement une instruction retardée au sein du code est appelée instruction de retard (*delay slot*). L'existence de retards implique que l'instruction qui suit un branchement ou un appel de routine (une procédure ou une fonction) est toujours exécutée. Par conséquent, si le branchement est conditionnel, il ne faut pas que l'instruction de retard modifie le registre contenant les codes de condition. C'est la raison pour laquelle la plupart des instructions existent en deux versions, une avec modification d'état, l'autre sans.

L'espace d'adressage de cette architecture est linéaire et paginé, c'est-à-dire que 2^{32} octets sont adressables individuellement. Les mots mémoire sont de 32 bits et doivent être alignés sur des limites de mots, c'est-à-dire que les adresses doivent être des multiples de 4. Enfin, la mémoire est de type big-endian, donc l'octet de poids faible d'un mot est à l'adresse la plus

haute du mot. SPARC est une architecture du type chargement/rangement (*Load/Store*) aussi appelée architecture de type registre-registre ce qui signifie que tous les calculs s'effectuent entre des registres et que les accès à la mémoire sont explicites et sont réalisés par des instructions dédiées. En effet, les instructions opératoires (arithmétiques et logiques) n'admettent que des registres ou des constantes signées codées sur 13 bits pour opérandes, mais en aucun cas une référence à la mémoire. Les seules instructions qui ont accès à la mémoire sont les instructions de chargement d'une zone référencée de la mémoire dans un registre (Load) et les instructions de rangement du contenu d'un registre dans une zone référencée de la mémoire (Store).

La plupart des architectures implémentent un concept d'appel entre les routines (fonctions ou procédures) avec une pile. L'état des variables locales, les paramètres et le pointeur d'instruction de la fonction appelante sont stockés sur une pile au moment d'un appel de fonction, afin de libérer les registres. La fonction appelée peut utiliser ses paramètres qui sont disponibles sur la pile. SPARC utilise une autre technique pour passer les paramètres à la fonction appelée et sauvegarder les variables locales : le système de fenêtrage des registres. L'architecture définit les registres servant à stocker les variables locales, rechercher les paramètres d'entrée et stocker les paramètres de sortie, respectivement les registres l0-l7, i0-i7 et o0-o7 ce qui constitue une fenêtre de 24 registres. En plus, chaque fonction peut accéder à 8 registres globaux g0-g7 (ces huit registres sont communs à l'ensemble des routines du programme, c'est à dire qu'ils sont accessibles par toutes les fonctions du programme) ce qui fait en tout un total de 32 registres logiques accessibles simultanément par une routine donnée. Le mécanisme de fenêtrage est recouvrant, donc les registres d'entrée associés à une routine appelée correspondent aux registres de sortie de la routine appelante ce qui assure un passage transparent des paramètres.

Evidemment, ce système de fenêtre permet de réduire de manière importante les accès à la pile (donc les accès à la mémoire) lors des appels de fonctions ce qui apporte un gain en temps d'exécution des programmes vu que les données sont disponibles directement dans les registres du processeur. La pile est utilisée seulement en dernier recours lorsqu'une exception est levée en cas de non disponibilité des registres qui est un cas extrême. Un registre interne de 5 bits, le pointeur de fenêtre courante (*Current Window Pointer* désigné par CWP) référence la fenêtre de registres courante. Cela signifie que nous pouvons manipuler jusqu'à 32 fenêtres à la fois, soit un total de 520 registres. La figure III-2 montre le passage d'une fenêtre à une autre à travers les appels assembleur (SAVE/RESTORE).

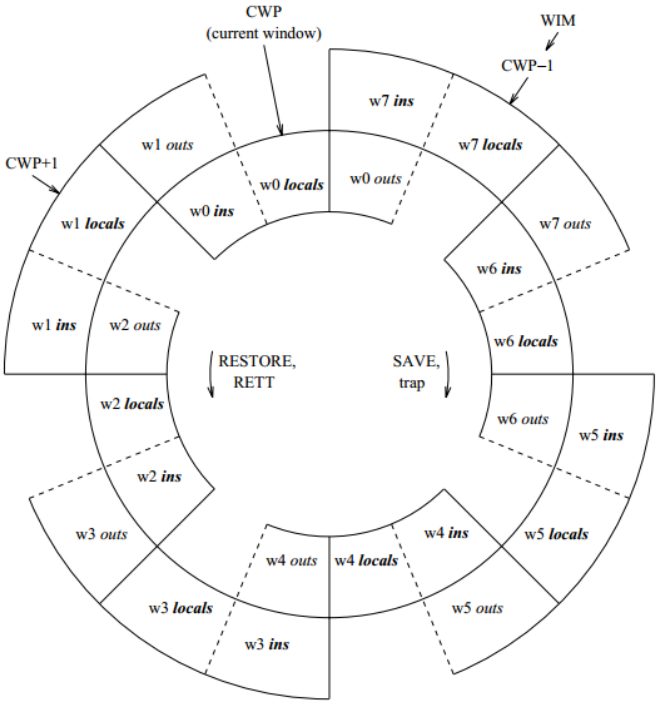


Figure III-2 : Fenêtrage des registres dans l'architecture SPARC v8 [SPARC. 92].

En outre, l'architecture SPARC v8 met à disposition du programmeur un système particulier de gestion d'exceptions sous forme de *traps*. Une *trap* peut être produite suite à une erreur lors de l'exécution d'une instruction dans le pipeline, peu importe à quel étage. C'est un transfert vectoriel (se faisant à travers une table des *traps*) vers le système d'exploitation. En effet, avant d'exécuter chaque instruction, le processeur vérifie si une requête d'exception ou une requête d'interruption est présente. Si tel est le cas, le processeur sélectionne la requête la plus prioritaire et produit la trap correspondante. L'adresse de la routine de la trap correspondante est stockée dans le registre TBR (*Trap Base Register*). Cette émission de *trap* est associée au système de fenêtrage précédent puisqu'elle va provoquer le passage dans la fenêtre inférieure si cela est possible. Dans cette nouvelle fenêtre, elle va pouvoir être gérée par le système d'exploitation, et causer ou non une exception dans l'exécution du code.

III.3.2. Le processeur LEON3

Le LEON3 est disponible comme un bloc réutilisable synthétisable libre de droits ("soft-core open source"), en téléchargement sous la forme d'une description en langage VHDL synthétisable. Ce cœur, conçu et maintenu par Cobham Gaisler [Gais. 13] sous contrat avec l'agence spatiale européenne (ESA), est placé sous licence LGPL. Le processeur appartient à la famille RISC 32 bits et a été certifié conforme à l'architecture SPARC v8. Le LEON3 a les caractéristiques suivantes :

- Pipeline de 7 étages.
- Une architecture Harvard avec deux caches (instructions/données), deux RAM (instructions/données), pilotées par une unité de gestion (MMU).
- Multiplieur et diviseur matériels (optionnels).

- Contrôleur d'interruption.
- Unité de debug et extensions multiprocesseur.

L'utilisation de ce processeur est en principe basée sur les interconnexions au travers d'un bus AMBA (*Advanced Microcontroller Bus Architecture*). Ce bus permet de faire un réseau de processeurs. En effet, le LEON3 supporte le Multi-processing symétrique (SMP- *Symmetric shared Memory MultiProcessor*) ou l'ajout de de nombreux périphériques. La figure III-3 présente le schéma bloc de ce processeur.

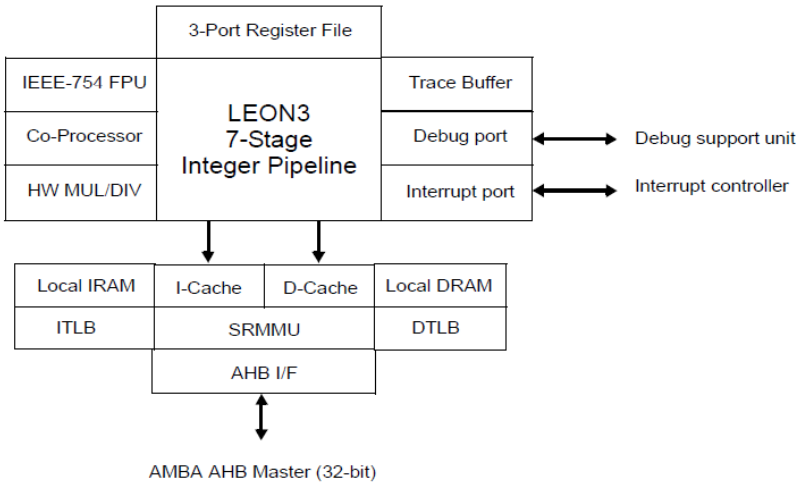


Figure III-3 : Schéma-bloc du processeur LEON3 [Gais. 13].

Le LEON3 est très configurable. De nombreux paramètres de configuration sont modifiables, comme par exemple : l'intégration d'un multiplieur-diviseur câblé, d'une unité pour nombres flottants ou d'un coprocesseur, la taille et la topologie des caches d'instructions et de données et le nombre de fenêtres de registres. Il est également possible de choisir une configuration multiprocesseur.

III.3.3. Unité entière du LEON3

L'unité entière est en mesure d'exécuter toutes les instructions SPARC v8. Elle est implémentée sous la forme d'un pipeline de sept étages :

- Etage *Fetch* : étage où la valeur de PC est générée et où les instructions sont lues depuis le cache d'instructions.
- Etage *Decode* : étage où les adresses de branchement et d'appel de fonction sont générées et où les instructions sont décodées.
- Etage *Register-Access* : étage où les valeurs des registres sont obtenues depuis le banc de registre ou grâce à un mécanisme de *forwarding*.
- Etage *Execute* : étage (disposant également d'un mécanisme de *forwarding*) où les adresses de lecture mémoire, de saut et de retour de fonction sont générées et les opérations arithmétiques, logiques ou de décalage sont réalisées. Cet étage est doté éventuellement d'un multiplieur/diviseur matériel.

- Etage *Memory* : étage où les lectures/écritures dans le cache de données sont effectuées.
- Etage *Exception* : étage où une trappe est, selon les conditions, activée ou non.
- Etage *Write-Back (WB)* : le résultat (s'il y en a un) est écrit dans le banc de registres.

La figure III-4 donne une description des chemins de données dans la microarchitecture de cette unité ainsi que les principaux registres implantés. Ces registres sont appelés registres internes de la microarchitecture ou tout simplement, registres du pipeline. Il existe aussi de nombreux éléments de mémorisation non visibles sur cette figure, qui correspondent à des registres de contrôle du pipeline.

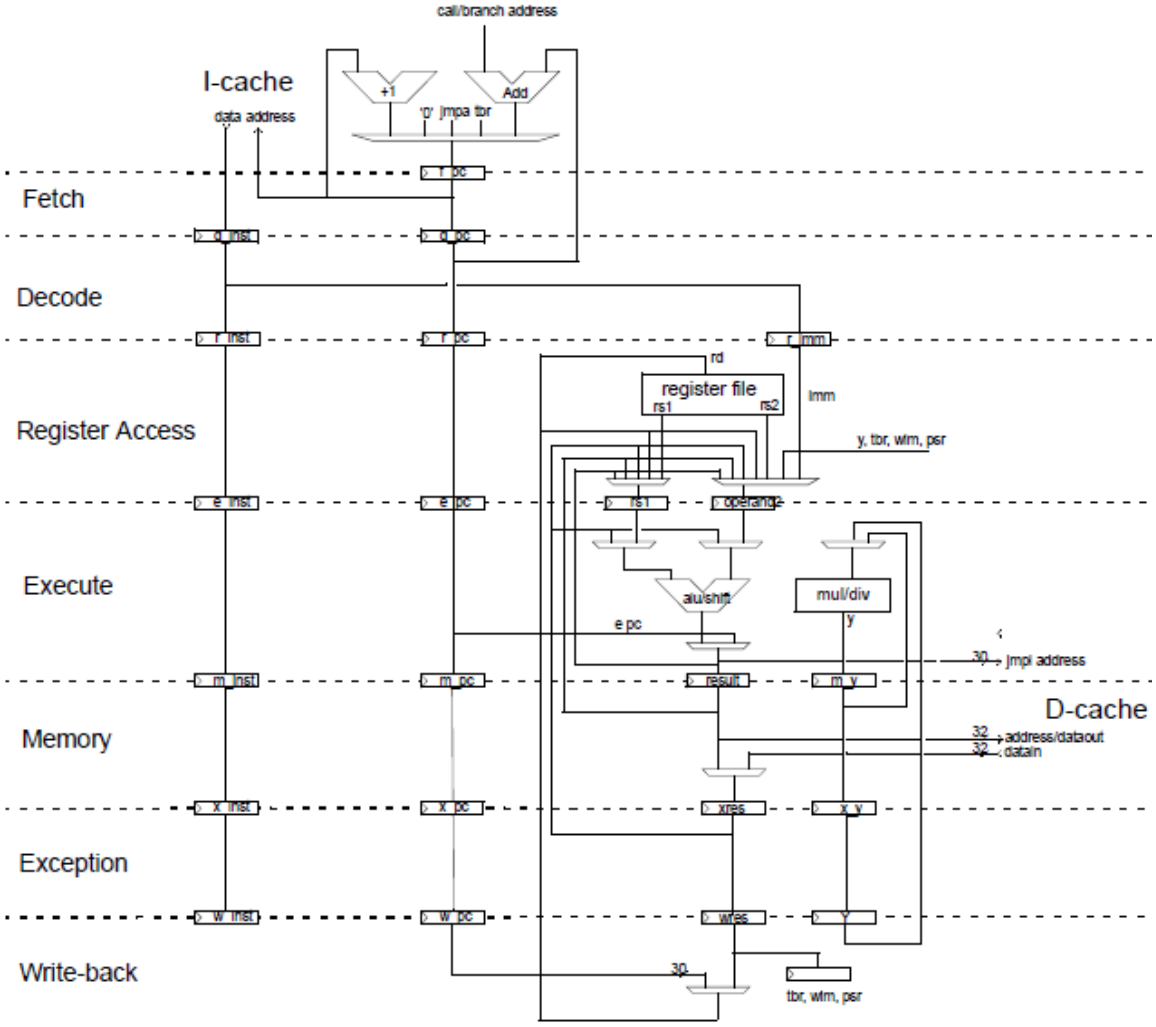


Figure III-4 : Pipeline de l'unité entière du LEON3 [Gais. 13].

Il est important de prendre en considération les détails suivants concernant le fonctionnement du pipeline de l'unité entière du LEON3.

- **Les mécanismes de *forwarding***

Ces mécanismes, appelés aussi mécanismes de *bypass*, représentent une solution matérielle pour les problèmes de dépendances de données (aléas de données) [Henn. 11]. Ils servent à la récupération des données depuis le pipeline, sans passer par le banc de registres. Comme la figure III-4 le montre, le pipeline du LEON3 dispose de deux mécanismes de *forwarding*:

- Le premier mécanisme renvoie une donnée depuis l'étage *Execute*, *Memory*, *Exception* ou *Write-Back* vers l'étage *Register-Access*.
- Le deuxième mécanisme renvoie une donnée depuis l'étage *Exception* vers l'étage *Execute*.

- **Instructions occupant plusieurs étages**

Il existe des instructions qui peuvent occuper plusieurs étages du pipeline au même cycle. Ces instructions contiennent des sous-instructions, par exemple, une instruction de type *STORE* (instruction de rangement) contient deux sous-instructions, la première pour générer l'adresse de l'emplacement mémoire et la deuxième pour générer la donnée à y écrire.

- **Situations de gel du pipeline**

Des situations de gel du pipeline (*pipeline stalls*) peuvent se produire lors de l'exécution d'une application à cause des défauts de cache (*cache miss*) ou d'une dépendance dans le pipeline (*pipeline hazards*).

- **Instructions annulées**

Une instruction en cours d'exécution peut être annulée tardivement à un étage du pipeline si le branchement qui la précède est pris. C'est la technique du bit d'annulation (*annul bit*) utilisée pour contourner les problèmes d'aléas de branchements.

- **Registres logiques / Registres physiques**

Le système de fenêtrage implique une notion de registres logiques et physiques. Le nombre de registres logiques est toujours 32, mais le nombre de registres physiques varie en fonction du nombre de fenêtres disponibles. A chaque registre logique correspond, à chaque instant, un et un seul registre physique dans le banc de registres. Plus de détails sur ce concept seront fournis dans la section suivante.

III.4. Mise en œuvre de l'approche : algorithme de prédiction de criticité

L'approche résumée dans la section III.2 va être appliquée sur le processeur LEON3. La configuration que nous utilisons dans le cadre de cette thèse est une configuration de base. En effet, nous n'utilisons ni l'unité de calcul flottant ni le coprocesseur ; nous nous intéressons uniquement à l'unité entière dont l'unité arithmétique et logique est le seul bloc de traitement

(l'unité de multiplication/division a été aussi désactivée). Le nombre de fenêtres de registres est égale à 8 ce qui fait, suivant l'architecture SPARC, un total de 136 registres physiques (8 fenêtres de 16 registres chacune plus les 8 registres globaux). Ces registres sont numérotés de 0 à 135 et forment le banc de registres. Notons par ailleurs que les options de configuration affectent les performances du processeur, mais pas la validité de l'approche.

III.4.1. Méthode d'analyse basée sur une trace de simulation

La méthode utilisée consiste à déterminer, à chaque cycle de l'exécution d'une application, l'état du pipeline de l'unité entière. Ensuite, en connaissant l'architecture interne de cette unité et les différents mécanismes implantés, nous pouvons conclure sur la criticité des registres. L'état du pipeline est obtenu grâce à une trace d'exécution. Cette trace peut être obtenue soit en ajoutant un process dans la description du processeur, soit en exploitant les fonctions des simulateurs permettant d'observer les signaux internes. A chaque cycle, des informations sont collectées sur le fonctionnement de chaque étage (de *Decode* à *Write-Back*). Pour un étage donné, les données enregistrées sont :

- L'instruction sous format hexadécimal (INST),
- L'adresse de l'instruction sous format hexadécimal (PC), chaque adresse correspondant à une et une seule instruction,
- Le numéro de la fenêtre utilisée (F) ; le nombre de fenêtres étant 8 dans notre configuration, ce numéro varie entre 0 et 7,
- La validité de chaque instruction (V). Pour ce champ, la valeur 1 montre que l'instruction est valide dans l'étage tandis que la valeur 0 indique que l'instruction a été annulée à ce niveau.

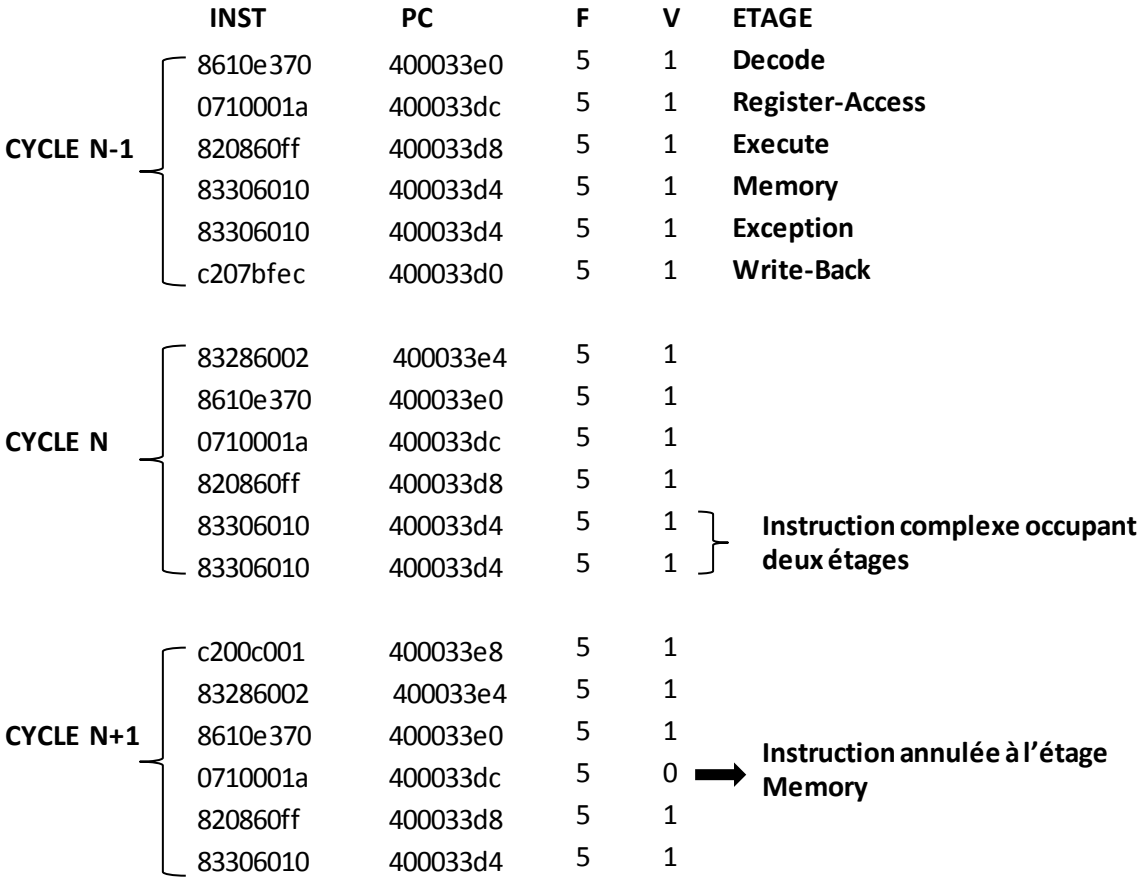


Figure III-5 : Extrait du fichier généré après simulation du LEON3.

La figure III-5 montre un exemple d'extrait de ce fichier qui est nommé "capture". La trace ne prend pas en compte l'étage *Fetch*. En effet, en se référant à l'architecture du pipeline (figure III-3), nous pouvons constater que l'étage *Fetch* dispose d'un seul registre qui est le compteur programme (PC) et qui représente l'adresse de l'instruction à chercher dans le cache d'instructions. Donc à ce stade, aucune instruction n'est disponible et le code de l'instruction à exécuter n'est disponible qu'à partir de l'étage *Decode*.

L'exécution d'un programme sur le LEON3 est articulée en trois étapes :

- Un boot matériel,
- Un boot logiciel, au cours duquel les registres sont initialisés.,
- L'exécution du programme principal, c'est à dire l'accès à la fonction "main" qui est le point d'entrée du programme et qui fait appel aux différentes fonctions développées.

Dans le cadre de notre travail, on s'intéresse au comportement des registres uniquement dans le programme principal. Un traitement a donc été appliqué sur la capture d'état pour ne garder que la partie intéressante de l'exécution. Ce traitement génère un fichier nommé "capture main".

III.4.2. Algorithme de calcul de criticité

Pour mettre en place la méthode décrite dans la section précédente, nous avons élaboré un algorithme permettant de calculer la criticité des différents registres généraux (registres du banc de registres) et des registres interne de l'unité entière (registres du pipeline) du processeur LEON3. Le fonctionnement global de cet algorithme est donné par la figure III-6.

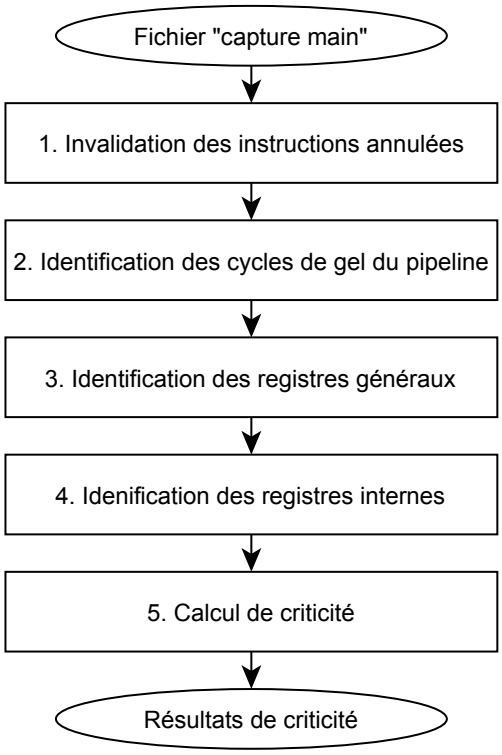


Figure III-6 : Algorithme de calcul de criticité des registres du LEON3.

Après la simulation du LEON3 et l'obtention de la trace "capture main", l'algorithme invalide les instructions qui ont été annulées tardivement et détermine les cycles où il y avait des situations de blocage dans le pipeline. Ensuite, il fait la conversion des instructions codées sous le format hexadécimal en code binaire. Puis, un traitement est effectué sur ce code binaire pour préciser les registres généraux et internes utilisés par les différentes instructions dans le pipeline. Enfin, le calcul de criticité repose sur les registres manipulés à chaque cycle et qui ont été déjà identifiés dans les étapes précédentes. Le fichier de sortie présente les pourcentages de criticité des différents registres. Ces étapes sont présentées en détail dans les sections suivantes.

III.4.2.1. Invalidation des instructions annulées

Cette étape consiste à identifier les instructions qui ont été invalidées après leur apparition dans le pipeline. En effet, le processeur peut charger des instructions dans le pipeline alors qu'elles n'auraient pas dû l'être pour obtenir des gains en performances [Henn. 11]. Une instruction peut très bien être invalidée dans l'étage *Memory*. Elle sera alors valide dans les étages *Decode*, *Register-Access* et *Execute* et invalide dans les étages suivants. Il faut donc aussi,

pour l'analyse de criticité, invalider cette instruction dans les étages précédents pour garder une trace d'exécution cohérente. La figure III-7 présente cet exemple de cas.

	INST	PC	F	V		INST	PC	F	V
	8610e370	400033e0	5	1		8610e370	400033e0	5	1
	0710001a	400033dc	5	**1**		0710001a	400033dc	5	**0**
CYCLE N-1	820860ff	400033d8	5	1		820860ff	400033d8	5	1
	83306010	400033d4	5	1		83306010	400033d4	5	1
	83306010	400033d4	5	1		83306010	400033d4	5	1
	c207bfec	400033d0	5	1		c207bfec	400033d0	5	1
	83286002	400033e4	5	1		83286002	400033e4	5	1
	8610e370	400033e0	5	1		8610e370	400033e0	5	1
CYCLE N	0710001a	400033dc	5	**1**	➡	0710001a	400033dc	5	**0**
	820860ff	400033d8	5	1		820860ff	400033d8	5	1
	83306010	400033d4	5	1		83306010	400033d4	5	1
	83306010	400033d4	5	1		83306010	400033d4	5	1
	c200c001	400033e8	5	1		c200c001	400033e8	5	1
	83286002	400033e4	5	1		83286002	400033e4	5	1
CYCLE N+1	8610e370	400033e0	5	1		8610e370	400033e0	5	1
	0710001a	400033dc	5	**0**		0710001a	400033dc	5	**0**
	820860ff	400033d8	5	1		820860ff	400033d8	5	1
	83306010	400033d4	5	1		83306010	400033d4	5	1

Invalidations

Figure III-7 : Invalidation des instructions annulées dans la trace de simulation.

A la fin de ce traitement, toutes les instructions qui ont été annulées pendant l'exécution sont facilement identifiées. Ces instructions ne sont pas prises en compte dans le calcul de criticité des registres puisque leur résultat n'est pas exploité par le programme.

III.4.2.2. Identification des cycles de gel de pipeline

Quand une instruction sort de l'étage *Write-Back*, si elle utilise un registre destination alors la valeur de celui-ci est mise à jour immédiatement dans le banc de registres. De même, lorsqu'une instruction quitte l'étage *Register-Access*, si elle a des registres sources alors les valeurs de ceux-ci sont disponibles immédiatement dans le pipeline (plus précisément dans les registres de l'étage *Execute*). Mais, quand il y a des situations de blocage de pipeline, les accès en lecture/écriture au banc de registres sont aussi bloqués. Ceci a un effet sur les instants de disponibilité des données au niveau du pipeline et du banc de registres. De ce fait, la criticité des différents registres est influencée. Par exemple, lors d'un gel du pipeline, si l'instruction bloquée à l'étage *Register-Access* utilise une valeur présente dans le banc de registres cela rend le registre contenant cette valeur plus critique puisqu'il servira à mémoriser cette valeur plus longtemps. La prise en compte des situations de gel s'avère donc indispensable pour effectuer une analyse précise de criticité.

Afin d'identifier les cycles de blocage, l'algorithme procède de la façon suivante : il compare à chaque fois les cycles (n) et (n+1), et si les instructions du cycle (n+1) sont identiques

à celles du cycle (n) alors il en déduit que le cycle (n) est un cycle de blocage. Un exemple de cas est illustré en figure III-8.

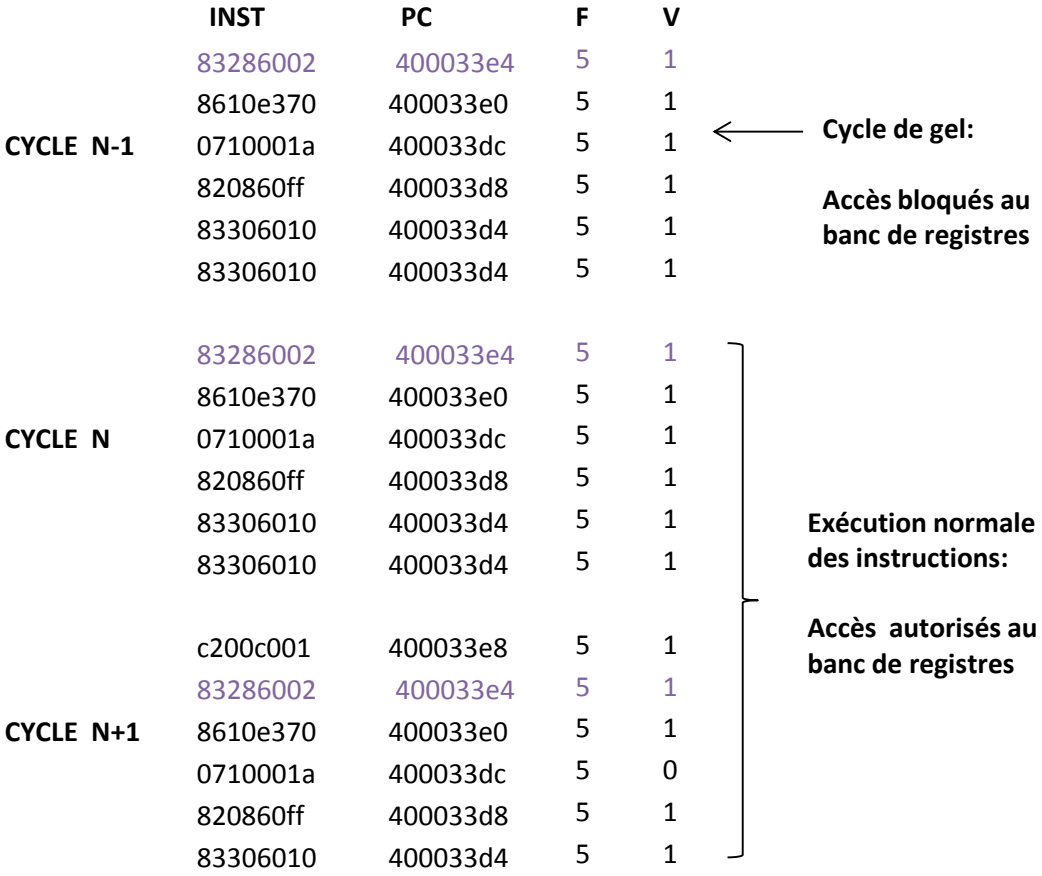


Figure III-8 : Détection d'un cycle de gel dans le pipeline.

III.4.2.3. Identification des registres généraux

Selon son format, une instruction dans le pipeline peut manipuler un ou plusieurs registres. Le premier traitement fait sur le code binaire permet d'identifier les registres logiques, or ce sont les registres physiques qui nous intéressent pour l'évaluation de criticité. Il est par la suite nécessaire de déterminer les registres physiques réellement employés. Les 8 registres logiques globaux (g0-g7) sont physiquement identiques tout au long de l'exécution d'un programme, c'est-à-dire qu'ils ne dépendent pas de la fenêtre de registres utilisée. Par contre, pour les autres registres, qui sont au nombre de 24, les registres physiques correspondants changent en fonction du nombre de fenêtres fixé lors de la configuration et de la valeur du registre "cwp". Le tableau III-1 donne la correspondance entre les registres logiques et physiques dans le cas de notre configuration (8 fenêtres).

Tableau III-1 : Correspondance entre les noms logiques des registres et leurs numéros dans le banc de registres en fonction du CWP.

		Registre physiques							
#	Registres	cwp = 0	cwp = 1	cwp = 2	cwp = 3	cwp = 4	cwp = 5	cwp = 6	cwp = 7
0	%g0	128	128	128	128	128	128	128	128
1	%g1	129	129	129	129	129	129	129	129
2	%g2	130	130	130	130	130	130	130	130
3	%g3	131	131	131	131	131	131	131	131
4	%g4	132	132	132	132	132	132	132	132
5	%g5	133	133	133	133	133	133	133	133
6	%g6	134	134	134	134	134	134	134	134
7	%g7	135	135	135	135	135	135	135	135
8	%o0	8	24	40	56	72	88	104	120
9	%o1	9	25	41	57	73	89	105	121
10	%o2	10	26	42	58	74	90	106	122
11	%o3	11	27	43	59	75	91	107	123
12	%o4	12	28	44	60	76	92	108	124
13	%o5	13	29	45	61	77	93	109	125
14	%o6 -	14	30	46	62	78	94	110	126
15	%o7	15	31	47	63	79	95	111	127
16	%l0	16	32	48	64	80	96	112	0
17	%l1	17	33	49	65	81	97	113	1
18	%l2	18	34	50	66	82	98	114	2
19	%l3	19	35	51	67	83	99	115	3
20	%l4	20	36	52	68	84	100	116	4
21	%l5	21	37	53	69	85	101	117	5
22	%l6	22	38	54	70	86	102	118	6
23	%l7	23	39	55	71	87	103	119	7
24	%i0	24	40	56	72	88	104	120	8
25	%i1	25	41	57	73	89	105	121	9
26	%i2	26	42	58	74	90	106	122	10
27	%i3	27	43	59	75	91	107	123	11
28	%i4	28	44	60	76	92	108	124	12
29	%i5	29	45	61	77	93	109	125	13
30	%i6 - %fp	30	46	62	78	94	110	126	14
31	%i7	31	47	63	79	95	111	127	15

III.4.2.4. Identification des registres internes

L'interprétation de l'opcode de l'instruction détermine la nature de celle-ci. Une fois l'instruction connue, elle est associée à un modèle prédéfini. Les modèles d'instructions ont été élaborés dans le but d'identifier, pour chaque instruction du jeu d'instructions SPARC (72 instructions), les registres effectivement utilisés à chaque étage du pipeline. Ces modèles ont été également validés par des simulations RTL du LEON3 exécutant des programmes écrits au niveau assembleur. La figure III-9 est un extrait du code source de l'algorithme de prédiction illustrant quelques exemples de modèles d'instructions.

```
case ADD:
sub_inst_models = new SubInstModel[nb_sub_inst];
sub_inst_models[0].setDecode(DECODE_INST, EMPTY, EMPTY);
sub_inst_models[0].setRegisterAccess(RFO_DATA1, RFO_DATA2, RD_CTRL);
sub_inst_models[0].setExecute(R_E_OP1, R_E_OP2, EMPTY);
sub_inst_models[0].setMemory(R_M_RESULT, EMPTY, EMPTY);
sub_inst_models[0].setException(R_X_RESULT, EMPTY, EMPTY);
sub_inst_models[0].setWriteBack(R_W_RESULT, EMPTY, EMPTY);
break;
case ADDccimm:
sub_inst_models = new SubInstModel[nb_sub_inst];
sub_inst_models[0].setDecode(DECODE_INST, EMPTY, EMPTY);
sub_inst_models[0].setRegisterAccess(RFO_DATA1, R_A_IMM, RD_CTRL);
sub_inst_models[0].setExecute(R_E_OP1, R_E_OP2, EMPTY);
sub_inst_models[0].setMemory(R_M_RESULT, EMPTY, R_M_ICC);
sub_inst_models[0].setException(R_X_RESULT, EMPTY, R_X_ICC);
sub_inst_models[0].setWriteBack(R_W_RESULT, EMPTY, R_W_S_ICC);
break;
case NOP:
sub_inst_models = new SubInstModel[nb_sub_inst];
sub_inst_models[0].setDecode(DECODE_INST, EMPTY, EMPTY);
sub_inst_models[0].setRegisterAccess(EMPTY, EMPTY, EMPTY);
sub_inst_models[0].setExecute(EMPTY, EMPTY, EMPTY);
sub_inst_models[0].setMemory(EMPTY, EMPTY, EMPTY);
sub_inst_models[0].setException(EMPTY, EMPTY, EMPTY);
sub_inst_models[0].setWriteBack(EMPTY, EMPTY, EMPTY);
break;
```

Figure III-9 : Modèles associés aux instructions "ADD", "ADDimm" et "NOP".

Les registres occupés dans le pipeline sont fortement liés au type d'instruction. Par exemple, les deux modèles relatifs aux instructions d'addition "ADD" et "ADDccimm" sont différents. Le mode d'adressage dans la première instruction est un adressage indirect tandis que la deuxième utilise un adressage immédiat et engendre une modification du registre d'état du processeur. Un autre exemple est celui de l'instruction NOP où seulement le registre *Decode_Inst*, situé à l'étage *Decode*, est utilisé.

III.4.2.5. Technique de calcul de criticité

L'objectif de la technique proposée consiste à évaluer les durées de vie des registres physiques contenus dans le banc de registres ou dans le pipeline.

III.4.2.5.1. Calcul de criticité des registres du banc de registres

Dans le but de faciliter la compréhension, nous allons focaliser notre étude sur un registre que l'on va nommer "R". Pour déterminer la durée de vie (D) de R, il a fallu fixer les conditions définissant sa criticité à un cycle d'exécution donné. Dans une instruction "I" valide et occupant un étage de pipeline, si le registre R est utilisé alors il peut être soit un registre source soit un registre destination.

- **Cas où R est un registre destination**

Dans le cas où R est présent uniquement comme un registre destination pour les instructions à un instant donné dans le pipeline aucun problème ne se pose car, quelques cycles plus tard, la valeur de R sera mise à jour et donc, si R contenait une valeur erronée, cette valeur sera écrasée après l'opération d'écriture. C'est le cas du registre numéro 44 de la figure III-10.

Etant donné que l'accès en écriture au banc de registres ne se fait qu'à l'étage *Write-Back*, l'algorithme observe seulement le registre destination de l'instruction située à cet étage. Ceci permet d'identifier les cycles de mise à jour des données dans les registres.

- **Cas où R est un registre source**

Un problème apparaît quand un accès en lecture de R est demandé, c'est-à-dire, lorsqu'une instruction I occupe l'étage *Register-Access* et utilise R comme un registre source. Si R ne représente pas un registre destination pour l'une des instructions situées dans les étages suivants alors R sera considéré comme critique car sa valeur est lue et doit être correcte pour le bon déroulement de l'exécution. Dans le cas contraire, R n'est pas critique parce que la valeur recherchée par "I" sera lue de manière anticipée depuis le pipeline sans passer par R (figure III-10).

		Numéro de registre à accéder en lecture (RS) ou en écriture (RD)		
ETAGE	**Instructions**	**RS1**	**RS2**	**RD**
Decode	I1	135	52	58
Register-Access	I2	127	49	69
Execute	I3	75	85	51
Memory	I4	57	44	127
Exception	I5	74	55	98
Write-Back	I6	28	134	44

Figure III-10 : Exemple de cas ou un registre source n'est pas critique.

Dans le cas de la figure III-10, l'instruction I2 sera située normalement au cycle suivant à l'étage *Execute* (sauf en cas de blocage) et n'aura pas besoin d'accéder au banc de registre pour lire la valeur du registre physique numéro 127. Cette valeur sera disponible automatiquement puisqu'elle est déjà prête à l'étage *Memory* (instruction I4). Ce transfert de données est assuré par le mécanisme de *forwarding* de l'étage *Register-Access*. En conséquence, si une faute atteint effectivement le registre 127, elle n'aura pas d'effets sur le résultat de l'instruction I2 à ce cycle.

Vu que l'accès en lecture au banc de registres ne s'effectue qu'à l'étage *Register-Access*, l'algorithme traite seulement les registres sources de l'instruction située à cet étage. Ceci permet, en conséquence, d'identifier les cycles où les registres généraux sont potentiellement exploités.

- **Absence totale ou partielle de R dans le pipeline**

Comme détaillé précédemment, l'algorithme cherche à analyser les registres utilisés par les instructions des étages *Register-Access* (pour les accès en lecture) et *Write-Back* (pour les accès en écriture). Si R ne figure pas dans ces étages, l'algorithme ne peut pas juger la criticité de R. Cette absence peut être partielle dans le sens où R peut être présent dans les opérandes des instructions présentes dans le pipeline, mais pour une instruction occupant un autre étage (*Decode*, *Execute*, *Memory* ou *Exception*). Elle peut aussi être totale au cas où R n'apparaît ni comme un registre source ni comme un registre destination pour les différentes instructions dans le pipeline.

L'algorithme doit être en mesure d'évaluer la criticité de R à tous les cycles, indépendamment de sa présence dans les étages *Register-Access* et *Write-Back*. Une partie de l'algorithme développé est chargée de traiter ce type de situation. En fait, il suffit d'examiner chaque nouvelle utilisation du registre R dans le pipeline. Un exemple concret peut expliquer simplement la situation. Supposons qu'à partir du cycle (i) le registre R n'est pas visible dans le pipeline. Au cycle (i+m), il se peut que R réapparaisse comme un registre source à l'étage *Register-Access*. Dans ces conditions, R est effectivement critique pendant les (m) cycles précédents car si sa valeur a été modifiée une donnée erronée sera transmise au cycle (i+m). En revanche, si R réapparait au cycle (i+m) comme un registre destination à l'étage Write-Back alors, durant les (m) cycles d'absence, R n'est pas critique parce que sa valeur est actualisée sans avoir été lue entre temps.

III.4.2.5.2. Calcul de criticité des registres internes

Pour les registres internes, nous nous sommes concentrés sur les registres les plus importants lors de l'exécution d'une instruction et qui sont disponibles dans les modèles d'instructions (section III.4.2.4). En effet, prendre en compte de manière détaillée chaque registre du pipeline augmenterait fortement la complexité des calculs sans améliorer de façon sensible la qualité globale de l'évaluation de la criticité. Les principaux registres analysés par l'algorithme sont :

- Le registre "D_INST" (32 bits) ; ce registre se trouve à l'étage *Decode*, il contient le code de l'instruction à décoder.
- Les registres "R_A_RFA1", "R_A_RFA2", "R_A_IMM" et "R_A_RD" ; ces registres sont situés à l'étage *Register-Access* et chaque registre joue un rôle bien déterminé :
    - R_A_RFA1 (5 bits) : indique le numéro du registre logique à accéder pour récupérer la valeur du premier opérande de l'instruction. Les 5 bits montrent qu'une instruction peut référencer 32 registres logiques ce qui est conforme avec la norme SPARC v8,

    - R_A_RFA2 (5 bits) : indique le numéro du registre logique à accéder pour récupérer la valeur du deuxième opérande de l'instruction,
    - R_A_IMM (32 bits) : contient la valeur immédiate en cas d'adressage immédiat,
    - R_A_RD (5 bits) : indique le numéro du registre logique de destination de l'instruction.
- Les registres "R_E_OP1", "R_E_OP2" et "R_E_SHCNT" de l'étage *Execute* :
    - R_E_OP1 (32 bits) : contient la valeur du premier opérande,
    - R_E_OP2 (32 bits) : contient la valeur du deuxième opérande ou bien la valeur immédiate en cas d'dressage immédiat,
    - R_E_SHCNT (5 bits) : ce registre est utilisé dans les opérations logiques de décalage.
- Les registres "R_M_RESULT" et "R_M_ICC" qui se trouvent à l'étage *Memory* :
    - R_M_RESULT (32 bits) : sert à stocker le résultat de l'instruction exécutée à l'étage précédent. Ce registre est utilisé ensuite pour des opérations de lecture ou d'écriture dans le cache des données.
    - R_M_ICC (4 bits) : ces 4 bits sont un sous registre du registre d'état du LEON3 et ils sont utilisés en cas de branchements conditionnels ou de traps conditionnelles.

En fonctionnement normal du pipeline, c'est–à-dire sans blocage, ces registres sont systématiquement mis à jour à chaque front d'horloge. Par contre, en cas de blocage, ils sont forcés à maintenir plus longtemps leur valeur, ce qui les rend plus critiques.

Le calcul de criticité des registres internes consiste à évaluer leur utilisation par les instructions valides dans le pipeline. L'algorithme vérifie, à chaque cycle, si le registre en question est utilisé ou pas suivant les modèles d'instructions. Si c'est le cas alors ce registre sera marqué comme critique à ce cycle. Le même principe d'évaluation est utilisé pour la plupart des registres sauf quelques cas particuliers que nous allons préciser ci-après.

- **Bits inutilisés dans le registre d'instruction D_INST**

Le registre D_INST est un registre de 32 bits, et n'importe quelle instruction, codée éventuellement sur 32 bits, passe par ce registre pour que le processeur puisse la décoder ultérieurement. Mais, dans certaines instructions, certains bits sont inutilisés donc si une faute transitoire survient sur l'un de ces bits elle ne sera pas activée ; elle est dite silencieuse dans ce cas.

Un exemple concret est celui de l'instruction NOP. La présence de cette instruction dans un programme est liée à la forte présence des branchements, des sauts et des appels de fonctions. En fait, dans certains cas lors de la compilation, le compilateur insère des instructions NOP après les instructions de branchement car il ne trouve pas d'instruction utile à placer après le branchement retardé. Une erreur sera générée seulement dans le cas où la faute affecte l'un des bits de l'opcode (la partie qui détermine la nature et le type de l'instruction à exécuter lors

du décodage). Dans le cas contraire, aucune erreur ne sera produite puisque le processeur a déjà compris qu'il s'agit d'une instruction NOP et qu'il ne va rien exécuter. De ce fait, quoi qu'il arrive aux autres bits, il n'y aura aucune erreur au niveau des résultats attendus de l'application exécutée. La figure III-11 présente l'exemple de l'instruction NOP.

Figure III-11 : Contenu du registre D_INST en cas d'instruction NOP.

L'algorithme de prédiction ne peut pas prévoir quel bit sera affecté lors d'une perturbation. La solution proposée est d'exclure ou d'inclure les instructions dont certains bits sont inutilisés selon la probabilité d'activation d'une faute. Pour l'instruction NOP, seulement 5 bits du registre d'instruction sont utilisés donc la probabilité qu'une faute de type SEU atteignant ce registre provoque une erreur est égale à (5/32) soit 15%. La probabilité étant faible, nous avons décidé d'exclure l'instruction NOP du calcul de criticité du registre D_INST. D'autres choix pourraient être faits, par exemple dans le cas des fautes multiples (cas des MBUs).

- **Mécanismes de *forwarding***

La criticité des registres situés à l'étage *Register-Access* et à l'étage *Execute* ne dépend pas seulement des situations de gel mais aussi des mécanismes de *forwarding*. Ces mécanismes ont un impact significatif sur la détermination des cycles où des données utiles sont stockées dans certains registres du pipeline. Ils sont implantés dans les chemins de données de l'unité entière et détectés par comparaison entre les différents registres source et destination des instructions occupant le pipeline.

Les registres R_A_RFA1 et R_A_RFA2 servent à identifier les registres sources de l'instruction à exécuter. Comme le montre la figure III-12, si le mécanisme de *forwarding* relatif à l'étage *Register-Access* est activé, cela signifie que le processeur va récupérer les données sans passer par le banc de registres et donc sans avoir recours au registre R_A_RFA1 ou/et R_A_RFA2 (suivant le cas). Une faute qui arrive sur ces registres n'aura aucun effet puisqu'ils ne seront pas utilisés par le processeur. Au cycle suivant la faute sera écrasée suite à une nouvelle écriture dans ces registres, la faute est donc silencieuse.

Les registres R_E_OP1 et R_E_OP2 sont les principaux points d'entrée de l'unité arithmétique et logique. Pareillement à l'étage *Register-Access*, si le mécanisme de *forwarding* de l'étage *Execute* est activé alors le processeur n'aura pas besoin d'utiliser le registre R_E_OP1 ou/et R_E_OP2 (suivant le cas). En conséquence, une faute dans ces registres est silencieuse.

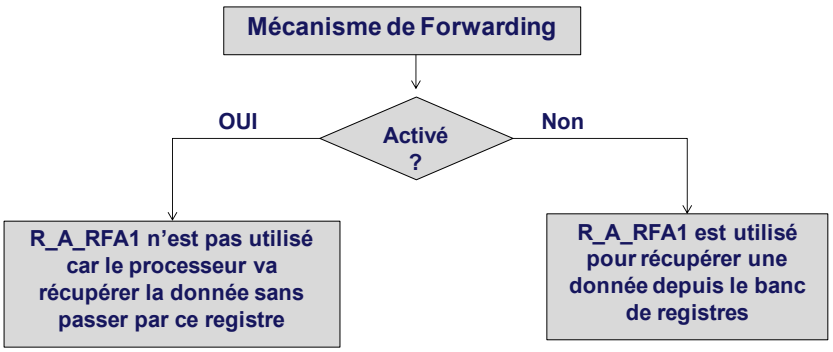


Figure III-12 : Impact du mécanisme de *forwarding* sur l'utilisation d'un registre interne.

En résumé, à un cycle donné, pour qu'un registre parmi ceux cités précédemment soit critique, il faut s'assurer que ce dernier n'est pas ignoré par un mécanisme de *forwarding* particulier.

III.5. Validation de l'approche par injection de fautes

Maintenant que l'algorithme de prédiction de criticité est développé, il est nécessaire d'évaluer son efficacité sur le plan pratique. Pour se mettre dans des situations réelles de perturbations, différentes campagnes d'injection de fautes ont été effectuées sur le LEON3 en phase d'exécution. Le but est de comparer les résultats de criticité obtenus par l'algorithme de prédiction avec ceux issus des campagnes d'injection.

III.5.1. Techniques d'injection de fautes utilisées

Pour injecter des fautes de multiplicité spatiale de 1 bit (SEU), nous avons utilisé deux techniques d'injection : l'injection par simulation et l'injection par émulation en utilisant la technique d'endo-reconfiguration sur FPGA. La première est plus simple et permet des analyses plus détaillées pour comprendre des cas particuliers ; la seconde permet d'accélérer les campagnes d'injection plus intensives et systématiques.

III.5.1.1. Injection de fautes par simulation

Cette technique consiste à utiliser les commandes du simulateur, lors de la simulation RTL du LEON3, afin d'injecter une faute en forçant un signal déterminé. Modelsim étant l'outil de simulation utilisé, un script TCL a été écrit dans le but d'assurer des injections de fautes pendant l'exécution du programme principal de l'application exécutée par le LEON3. Pour ce faire, il est nécessaire de préciser les cibles des injections. Une cible est caractérisée par :

- Le numéro de cycle d'injection,
- La référence du registre dans lequel on souhaite faire l'injection,
- Le numéro du bit à modifier dans le registre (un seul point mémoire est concerné pour une injection de SEU).

Les cibles d'injection sont générées grâce à un générateur de nombres pseudo-aléatoires appelé algorithme de Mersenne-Twister [MT. 98]. La figure III-13 représente le flot d'une compagne d'injection par simulation.

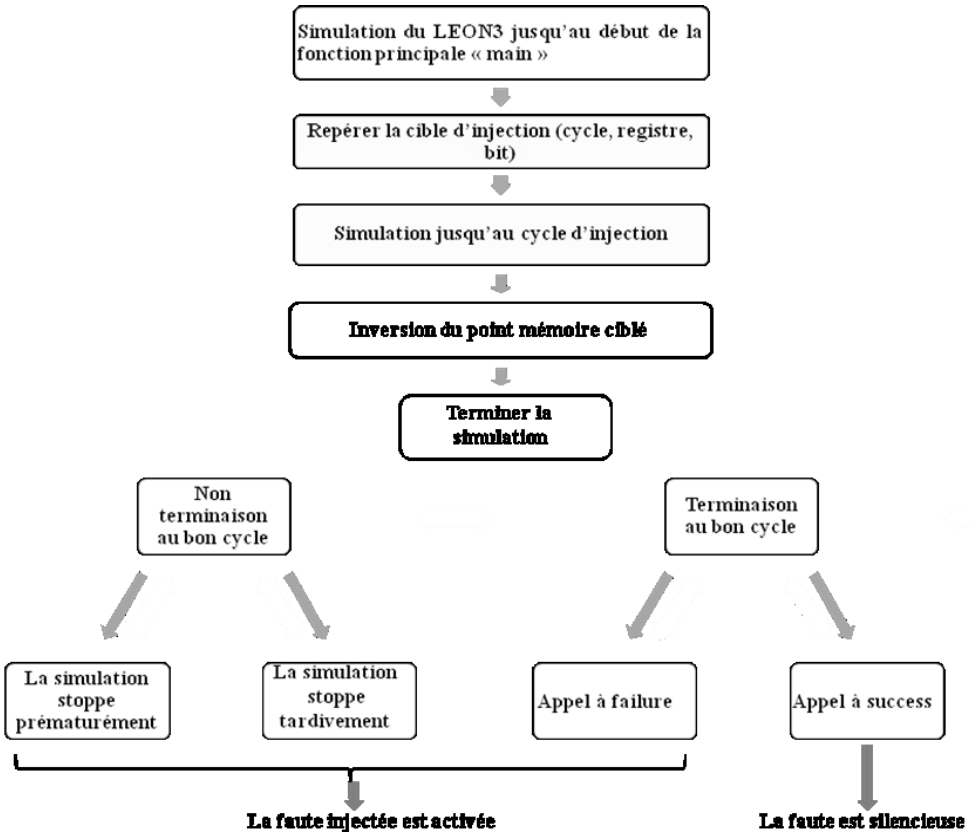


Figure III-13 : Flot d'une compagne d'injection par simulation.

Le script d'injections est une liste d'actions qui devront être opérées par Modelsim, dans le but de tester la résistance aux fautes du LEON3. Il s'agit de simuler le LEON3 jusqu'au début de la fonction principale du programme (fonction "main"), de repérer la cible d'injection et d'inverser le bit dans le registre concerné puis de terminer la simulation. Une fois que la simulation est terminée, nous testons l'instant d'arrêt, ainsi que la pertinence des données. Si l'application se termine au bon cycle, une comparaison du résultat obtenu avec une valeur de référence est effectuée pour déterminer si le programme s'est bien déroulé. Si le résultat est conforme aux attentes, le programme fait un appel à la fonction "success()" ; dans le cas contraire, c'est la fonction "failure()" qui est appelée. Ces deux fonctions ont été ajoutées au code source de l'application pour distinguer les deux cas.

III.5.1.2. Injection de fautes par endo-reconfiguration

L'injection par simulation nécessite des temps expérimentaux très élevés dans le cas du LEON3 : la complexité du modèle VHDL du processeur implique un temps de simulation élevé. Afin d'accélérer les campagnes d'injection de fautes, nous avons eu recours à la technique d'injection par endo-reconfiguration. Comme indiqué dans le chapitre précédent, une telle technique est basée sur la reconfiguration dynamique partielle de certains FPGAs. Pour

appliquer cette technique, un prototypage du LEON3 a été mis en œuvre sur un FPGA Virtex V de Xilinx dont la référence est "LX110TXUPV5" [WW. 2].

L'environnement mis au point pour mener nos expériences utilise un processeur embarqué, à savoir un Microblaze, pour gérer la campagne d'injection de fautes et l'IP ICAP (propriété de Xilinx) comme port de reconfiguration pour effectuer l'endo-reconfiguration. Cet environnement permet l'injection d'inversions de bits uniques ou multiples dans la configuration et nous permet aussi de suivre le comportement d'un SEU dans les bascules utilisateur qui se trouvent dans les blocs logiques de configuration (CLB). L'injection de fautes peut être déclenchée à tout moment pendant l'exécution d'une application sur le FPGA. Pour pouvoir injecter dans les registres du LEON3, il faut reconfigurer les bascules le composant. La figure III-14 présente le flot général détaillé.

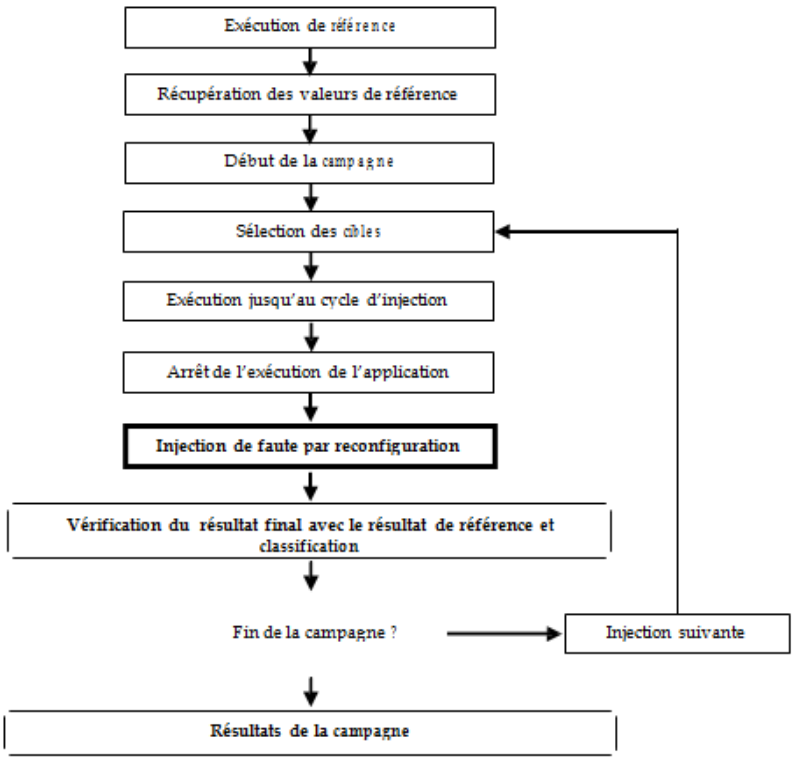


Figure III-14 : Flot général détaillé d'injection par endo-reconfiguration [Benj. 13].

Tout d'abord une exécution de l'application sans injection de fautes sert à enregistrer les valeurs de référence qui caractérisent le bon fonctionnement du programme. Ces valeurs vont permettre de classifier les résultats des injections. Ensuite, la campagne d'injection commence avec la sélection aléatoire du cycle et des cibles. L'application est exécutée jusqu'au cycle d'injection. A l'instant d'injection, le circuit analysé est mis en pause et l'injection de fautes est effectuée puis le circuit reprend son fonctionnement jusqu'à la fin de l'application. Ces étapes sont réitérées jusqu'à la fin de la campagne d'injection.

III.5.2. Analyse des résultats

Les applications exécutées lors de nos expériences proviennent de la suite MiBench [Mibe. 01] et sont décrites dans le tableau III-2. Ces applications ont été compilées à l'aide d'un cross-compilateur SPARC v8 (GCC V4.2.4) avec la macro-option -O2 par défaut.

Tableau III-2 : Présentation des applications utilisées.

Benchmark	Description
AES	Fonction standard de chiffrement/déchiffrement.
JPEG	Algorithme de compression/décompression d'images.
FFT	Transformée de Fourier rapide sur un tableau de données.
CRC32	Contrôle de redondance cyclique 32 bits sur un fichier.
SHA	Fonction de hachage cryptographique.

III.5.2.1. Comparaison des résultats de criticité

Des campagnes statistiques d'injection de fautes ont été réalisées dans les différents registres du LEON3 pendant l'exécution de chaque benchmark. Le nombre de fautes injectées est calculé, selon les études effectuées dans [Leve. 09], pour une précision avec 5% de marge d'erreur pour un taux de confiance de 95%. D'un autre côté, pour les différents benchmarks, nous avons lancé le programme de prédiction sur la trace obtenue après simulation du LEON3. Les figures III-15 et III-16 présentent un extrait des résultats de criticité obtenus par les deux traitements.

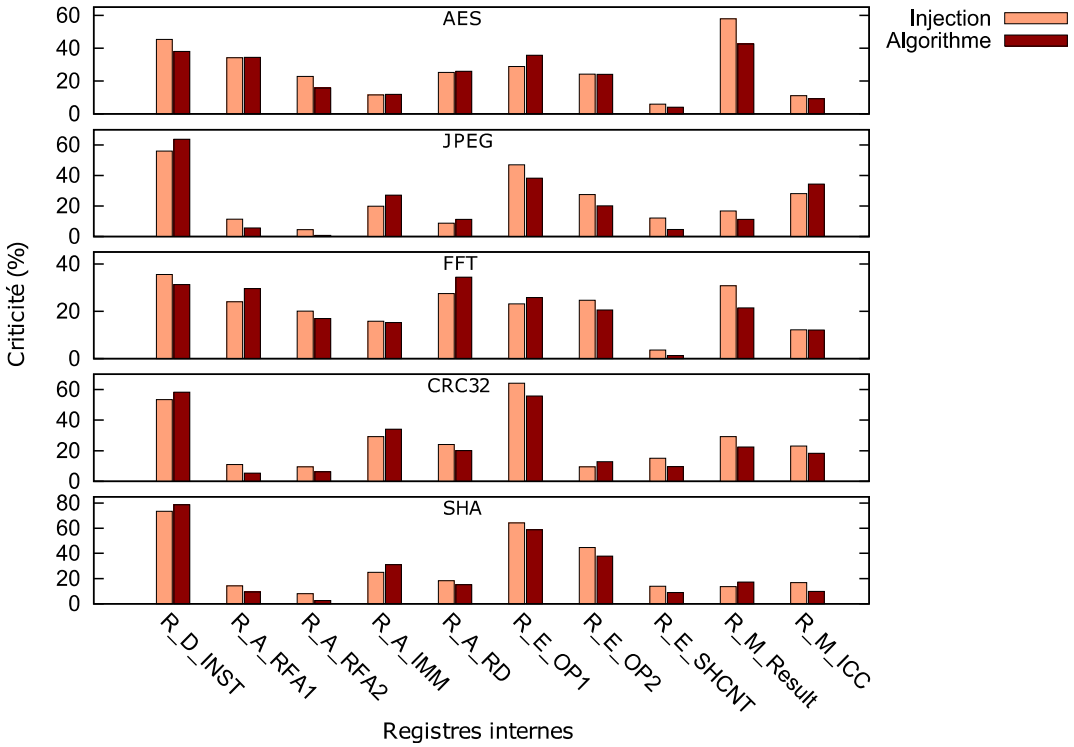


Figure III-15 : Comparaison des résultats de criticité des registres du pipeline.

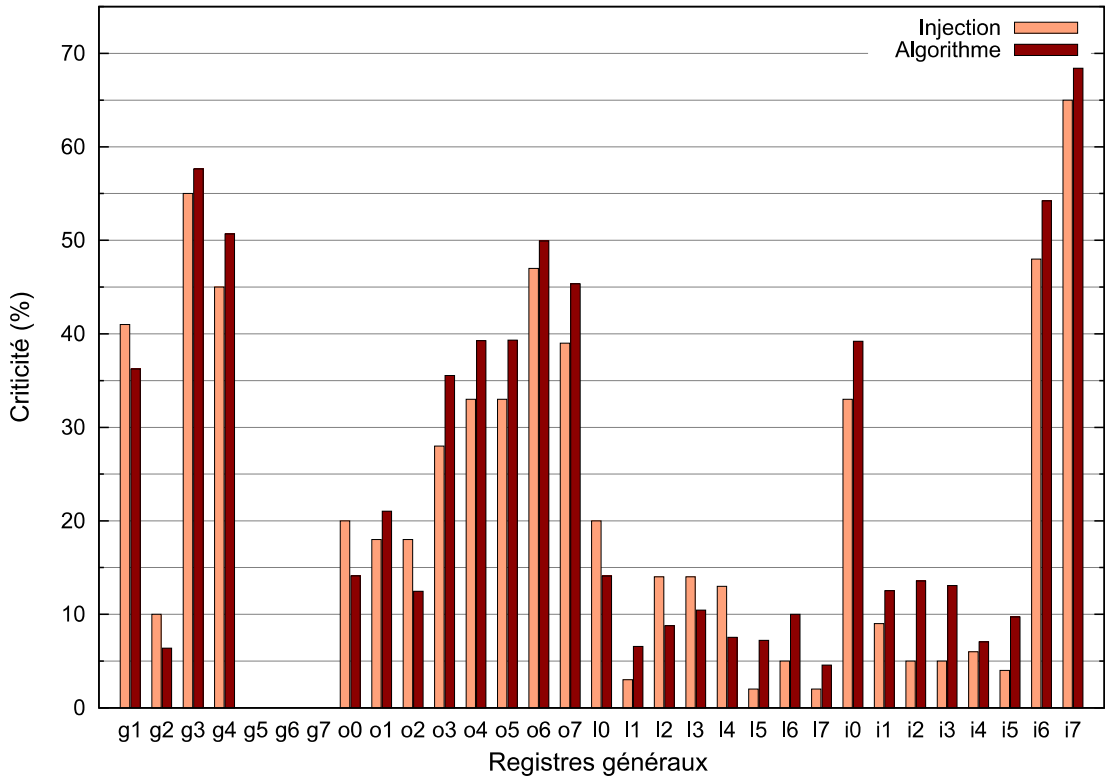


Figure III-16 : Comparaison des résultats de criticité des registres du banc de registres (CRC32).

Nous pouvons constater que les résultats donnés par l'algorithme sont bien en accord avec les résultats d'injection et respectent la marge d'erreur fixée. Cela est aussi confirmé par la figure III-17 visualisant les caractéristiques des distributions de criticité selon la méthode d'évaluation utilisée.

La criticité des registres logiques est déduite à partir des valeurs de criticité des registres physiques et du pourcentage d'utilisation de chaque fenêtre de registres pendant la simulation. Nous n'avons pas effectué des injections dans le registre %g0 puisqu'il contient toujours la valeur 0 (connexion permanente à la masse). Ce dernier est donc insensible aux fautes transitoires. La figure III-16 montre les résultats obtenus pour l'application CRC32 ; ceux obtenus pour les autres applications conduisent aux mêmes conclusions.

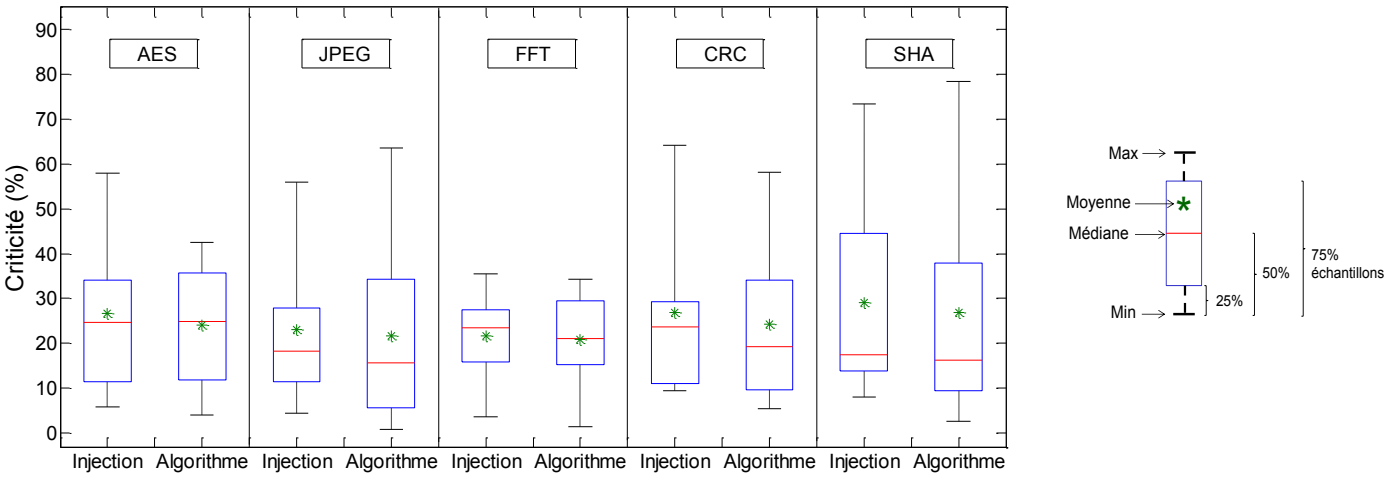


Figure III-17 : Comparaison des distributions de criticité des registres internes selon la méthode d'évaluation (Injection/Algorithme).

III.5.2.2. Identification des registres critiques

Dans le cadre d'une protection sélective, il peut être intéressant de protéger seulement un nombre limité de registres. Il est alors nécessaire d'identifier les registres ayant la plus forte criticité. La figure III-18 montre les registres internes les plus critiques identifiés à la fois par l'algorithme et par les injections de fautes, pour les cinq programmes d'application et pour des taux de protection allant de 10% à 60%.

Comme illustré par ces graphes, les registres internes sélectionnés par l'algorithme sont principalement les mêmes que ceux identifiés par les campagnes d'injection. Pour quelques exemples et certains niveaux de protection, nous avons observé une différence d'un registre entre les deux techniques. Mais, quoi qu'il en soit, cette différence est inférieure à la marge d'erreur des campagnes statistiques d'injection. Il est donc impossible de déterminer lequel des deux résultats est correct. Comme dans le cas précédent, la comparaison est montrée en figure III-19 pour CRC32 dans le cas du banc de registres et là encore les groupes identifiés par les deux approches sont très semblables.

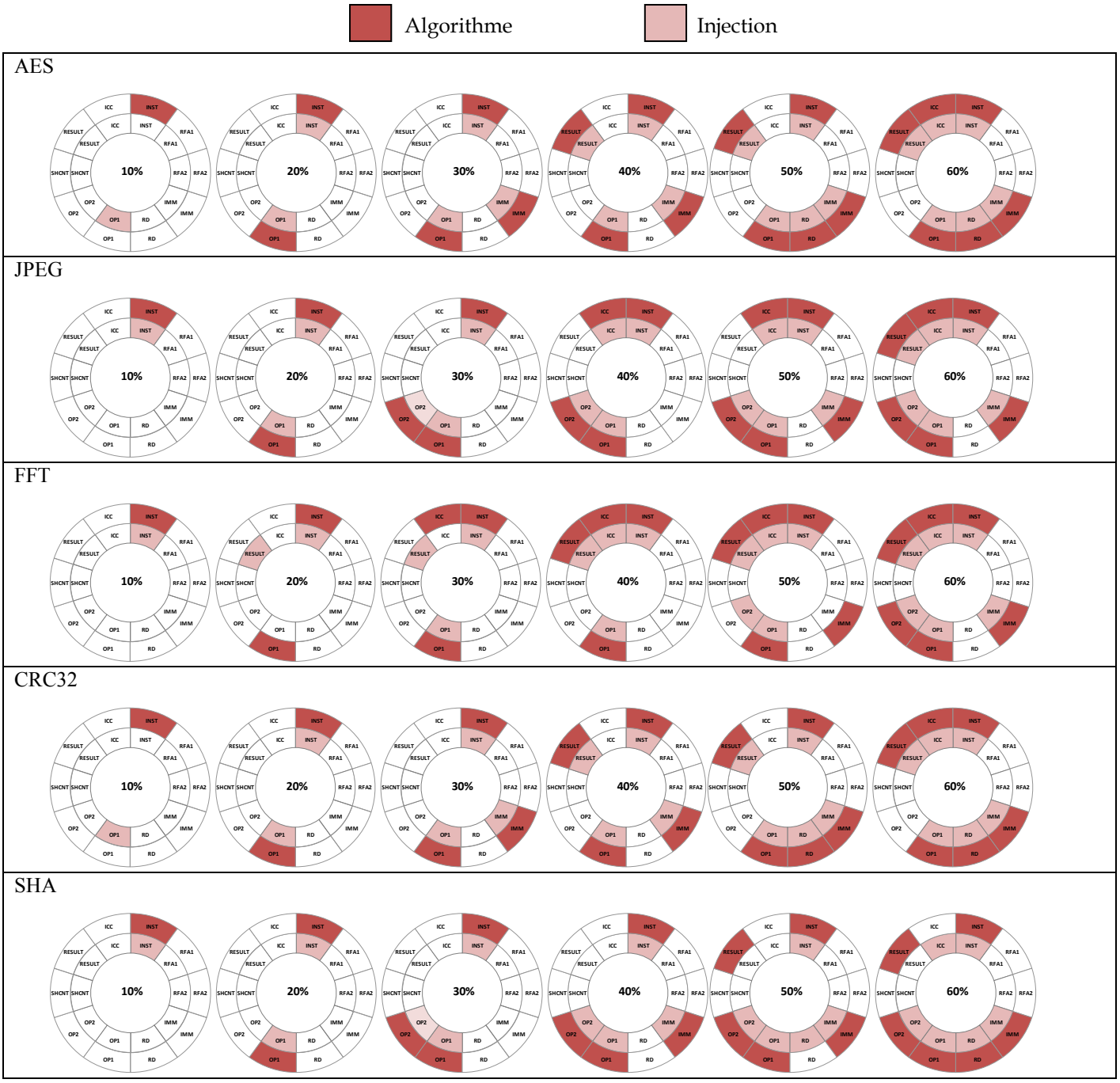


Figure III-18 : Registres internes critiques identifiés par l'algorithme et la technique d'injection.

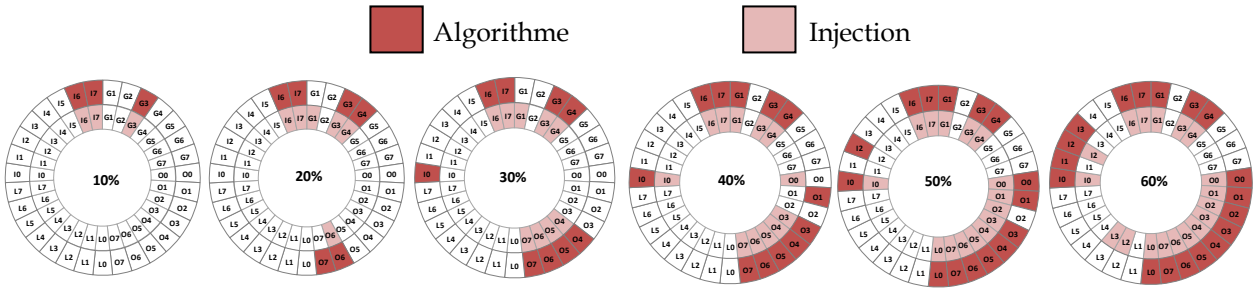


Figure III-19 : Registres généraux critiques identifiés par l'algorithme et la technique d'injection (CRC32).

III.6. Comparaison avec des évaluations de criticité obtenues lors de la compilation

III.6.1. Identification des registres critiques

L'approche développée dans [Berg. 10] a déjà été présentée dans le chapitre 2 (section II.3.2.1). Celle-ci repose sur des métriques calculées lors de la phase de compilation grâce à une version modifiée du cross-compilateur SPARC, ce qui permet d'obtenir rapidement des informations sur la criticité des registres utilisés dans le code généré. Cette approche a été appliquée sur le processeur LEON3 avec la même configuration que celle utilisée dans le cadre de notre étude. En vue de comparer l'algorithme de prédiction avec cette approche, les applications (tableau III-2) ont été compilées en utilisant la même option (-O2). La figure III-20 illustre, pour deux exemples d'application, les registres du banc de registres étant identifiés comme les plus critiques par les deux approches, en fonction du degré de protection visé.

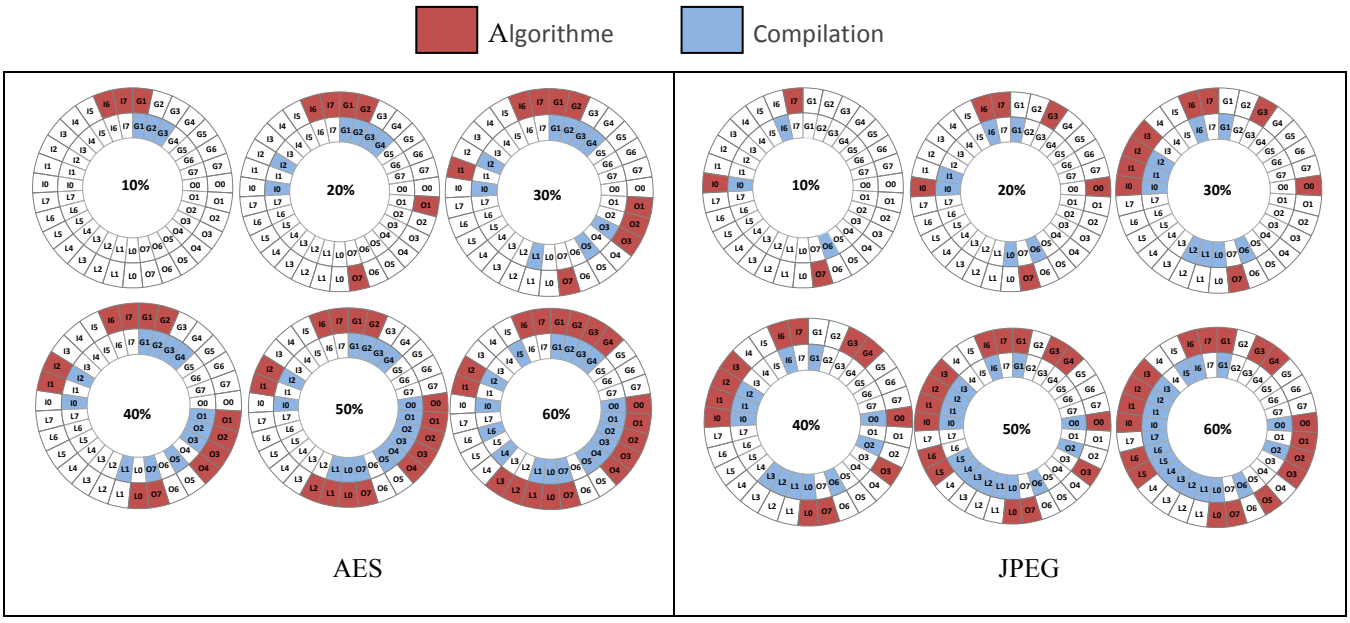


Figure III-20 : Registres critiques identifiés par l'algorithme et l'approche de compilation.

Les registres nominés par les deux techniques sont souvent assez différents. Cependant, il a été montré dans la section III.5 que l'algorithme de prédiction identifie sensiblement les mêmes registres que l'injection de fautes. Cela signifie que les évaluations de criticité lors de la phase de compilation ne sont pas très fiables si elles sont utilisées pour identifier les registres les plus critiques. En effet, contrairement à notre approche analytique, l'analyse statique du code lors de la compilation ne tient pas compte de plusieurs aspects du LEON3 (impact des registres du pipeline, situations de gel, etc.). La durée de vie réelle d'un registre architectural (comme ceux du banc de registres) est en fait répartie entre plusieurs registres physiques, la plupart d'entre eux étant des registres internes de la microarchitecture qui ne sont pas directement visibles par le programmeur, ni vraiment pris en compte par le compilateur. De plus, l'existence, dans le cas de l'architecture SPARC, d'un fenêtrage de registres, complique encore le lien entre la criticité des registres logiques et celle des registres physiques. Tout cela est probablement à l'origine de la non-concordance des résultats.

III.6.2. Impact des options de compilation sur la criticité

Compte-tenu des différences sur les évaluations de criticité, il nous a semblé utile de revisiter certaines évaluations qui avaient été réalisées avec les métriques obtenues en phase de compilation.

La deuxième partie des expérimentations vise donc à comparer la vulnérabilité de plusieurs versions du logiciel embarqué. Le but ici n'est pas d'identifier les registres les plus critiques, mais de comparer la robustesse intrinsèque de plusieurs implémentations de la même application. L'analyse de criticité lors de la compilation avait montré que les options de compilation ont un impact significatif sur la robustesse et que les options d'optimisation classiques avaient en général un effet négatif sur la durée de vie des variables, au moins pour les variables stockées dans le banc de registres ou dans la mémoire. Des expériences ont donc été effectuées sur la base de l'algorithme de prédiction pour confirmer ou infirmer cette constatation. Les résultats sont présentés dans le tableau III-3. Pour chaque application, cinq versions ont été générées en utilisant les macro-options disponibles dans le compilateur GCC pour l'optimisation du code. Chaque version est classée suivant son niveau de criticité globale (de 1 pour la plus critique à 5 pour la moins critique) à la fois par rapport aux registres généraux et aux registres internes de la microarchitecture.

Tableau III-3 : Ordres de criticité en fonction des macro-options de compilation.

Benchmark	Ordre de criticité (banc de registres / registres internes)				
	-O0	*-O1*	*-O2*	*-O3*	*-Os*
AES	5 / 5	4 / 4	1 / 3	3 / 1	2 / 2
CRC	5 / 5	2 / 3	1 / 1	4 / 2	3 / 4
FFT	3 / 3	2 / 2	5 / 5	4 / 4	1 / 1
JPEG	5 / 5	2 / 4	3 / 3	1 / 1	4 / 2
SHA	5 / 5	3 / 4	2 / 2	1 / 1	4 / 3

Pour la plupart des applications, la macro-option O0 (sans optimisations) s'est avérée la plus robuste et conduit à une criticité inférieure à celle fournie par les autres options. Ceci est bien cohérent avec la conclusion antérieure. Les résultats obtenus avec l'approche analytique prouvent également que les optimisations ont un impact négatif sur la criticité des registres internes du pipeline, ce qui n'avait pas pu être évalué lors de l'étude précédente. Naturellement, l'utilisation de O0 implique des coûts en termes de performances et/ou taille du code. Mais si l'objectif principal est de réduire la sensibilité des registres, O0 reste la meilleure option pour la plupart des exemples analysés. Il reste toutefois le contre-exemple FFT, pour lequel les optimisations sont globalement profitables, probablement à cause de la gestion de données à l'intérieur des boucles de calcul de la FFT. Une justification précise de ce comportement demanderait une analyse approfondie du code compilé, ce qui est en dehors du cadre de ce travail. Le tableau III-4 indique, pour chaque version des applications étudiées, le pourcentage de gel du pipeline (*pipeline stalls)*. Ce pourcentage est calculé par l'algorithme de prédiction comme étant le rapport entre le nombre de cycles de gel et le nombre de cycles total. Nous constatons que l'option O0 mène au nombre minimum de situations de blocage. Cela a un impact sur la durée de vie des informations dans tous les registres et, par conséquent, sur la robustesse intrinsèque de l'application.

Tableau III-4 : Pourcentages de gel du pipeline des différentes versions de logiciel.

Benchmark	**Options de compilation**				
	-O0	***-O1***	***-O2***	***-O3***	***-Os***
AES	32.47%	58.82%	60.01%	61.23%	60.11%
CRC	17.42%	31.09%	32.94%	37.10%	31.50%
FFT	30.81%	31.68%	44.43%	32.52%	31.21%
JPEG	21.32%	22.29%	21.15%	27.22%	21.34%
SHA	9.90%	12.28%	13.51%	21.32%	13.34%

Une autre explication du meilleur niveau de robustesse assuré par l'option O0 est le nombre élevé d'accès en écriture au banc de registres pendant l'exécution des programmes. Pour l'application JPEG, la figure III-21 représente la différence entre les cinq options en nombre de lectures/écritures effectuées dans le banc de registres. Le nombre très élevé des opérations d'écriture par rapport aux opérations de lecture avec l'option O0 induit un grand nombre de rafraîchissements des registres qui annulent certaines erreurs potentielles avant qu'elles n'impactent les calculs. Les mêmes caractéristiques ont été observées pour les autres applications.

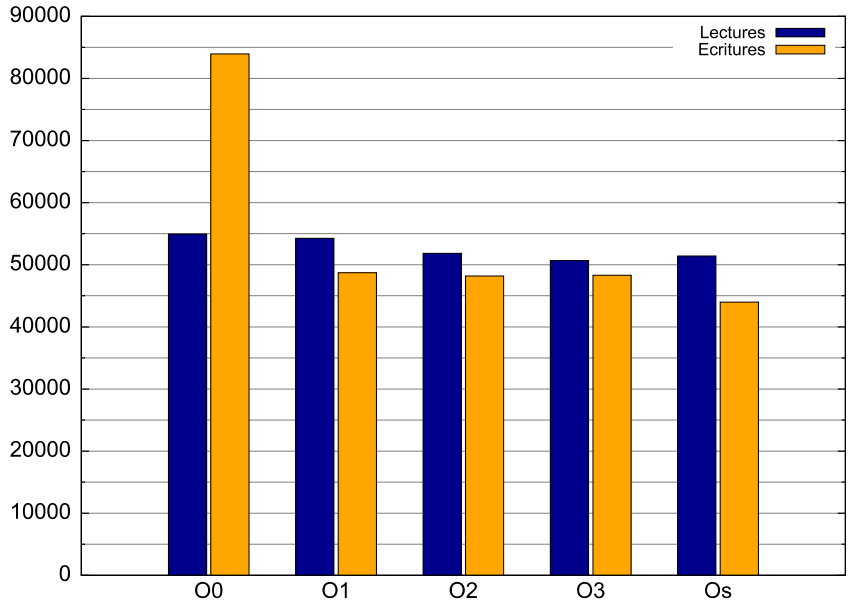


Figure III-21 : Nombre de lectures/écritures dans le banc de registres en fonction des macro-options pour l'application JPEG.

III.6.3. Comparaison des durées des expérimentations et synthèse

Nous avons comparé les résultats de l'algorithme de prédiction avec ceux obtenus par la technique d'injection de fautes en utilisant l'endo-reconfiguration et par l'analyse statique lors de la compilation. Le tableau III-5 illustre le temps requis par les trois approches pour réaliser l'analyse. La durée de la simulation est incluse dans le temps indiqué pour l'exécution de notre algorithme analytique. La machine sur laquelle nous avons effectué nos expérimentations est dotée d'un processeur Intel Dual Core cadencé à 3,16 GHz avec 4 Go de mémoire vive.

Tableau III-5 : Temps relatifs nécessaire pour chaque approche.

Benchmark	I-Injection	II-Algorithme	III-Compilation	Facteur d'accélération (II/I)	Facteur d'accélération (III/II)
AES	670 min	15 min	14 ns	45	64. 10^9
JPEG	990 min	18 min	114 ns	55	9. 10^9
FFT	820 min	17 min	5 ns	48	204. 10^9
CRC32	380 min	10 min	2 ns	38	300. 10^9
SHA	690 min	14 min	5 ns	49	168. 10^9

Nous remarquons que l'approche analytique réduit énormément les durées d'analyse par rapport à l'injection de fautes, même réalisée par émulation, tout en fournissant des résultats de criticité assez proches (section III.5.2). Il est à mentionner que le temps de mise en place de la plateforme d'injection (synthèse du LEON3, implémentation sur la carte, etc.) n'est pas pris en compte, ce qui confirme encore davantage la rapidité de l'algorithme de prédiction par rapport aux injections. Le facteur d'accélération est d'environ 50 ; ceci rend possible des évaluations itératives sur différentes versions d'une même application (section III.6.2), même

nettement plus complexes que les exemples cités ici. Les résultats montrent également que la durée d'évaluation dépend de la complexité de l'application. Pour l'algorithme de prédiction, cette durée varie légèrement. En revanche, pour les injections, les temps expérimentaux augmentent rapidement, non seulement en raison de la complexité des applications, mais aussi à cause du nombre de fautes à injecter.

En termes de rapidité, les ordres de grandeur de l'approche analytique et de celle basée sur la compilation sont clairement très différents. Les métriques mesurées pendant la compilation sont suffisantes pour une première analyse visant à comparer rapidement la robustesse de plusieurs versions d'un logiciel embarqué. Ceci donne la possibilité aux développeurs d'améliorer leur logiciel dès les premières étapes de développement du système. Cependant, comme nous l'avons vu, ces métriques ne sont pas suffisamment précises pour identifier les registres les plus critiques ou donner une bonne quantification de la robustesse. Bien que les injections de fautes permettent de faire des analyses beaucoup plus fines, elles peuvent nécessiter des temps prohibitifs même en exploitant la vitesse de l'émulation. Notre approche offre, en conséquence, le meilleur compromis entre l'efficacité et la rapidité des évaluations de criticité (tableau III-6).

Tableau III-6 : Comparaison des trois approches d'évaluation de robustesse.

Approches	Précision des évaluations	Rapidité d'analyse
Injection de fautes (émulation)	✔ ✔	✘
Approche analytique (algorithme de prédiction)	✔	✔
Evaluations lors de la compilation	✘	✔ ✔

III.7. Effets de quelques caractéristiques architecturales

Dans cette section, nous allons tirer profit de notre algorithme pour étudier les conséquences de la structure matérielle du processeur sur la criticité d'une application. Nous profitons du fait que le LEON3 est hautement configurable pour modifier chaque fois quelques caractéristiques architecturales et observer les conséquences que cela engendre. Nous analysons essentiellement l'impact de la taille du banc de registres et des différentes configurations des mémoires caches sur :

- La criticité globale mesurée en tant que la somme des cycles critiques de tous les registres,
- La durée d'exécution totale (en nombre de cycles),
- Le pourcentage de gel du pipeline,
- Le nombre total des accès mémoire. L'algorithme évalue ce paramètre en comptant le nombre d'occurrences des instructions *Load/Store* dans la trace de simulation (vu que l'accès à la mémoire se fait uniquement via ces instructions),.
- La marge entre le nombre d'écritures et de lectures par rapport au nombre total d'accès au banc de registres.

Pour chaque configuration du LEON3, une simulation fonctionnelle est effectuée afin d'enregistrer les traces nécessaires pour le lancement de notre programme de prédiction. Dans la suite, nous présentons les résultats relatifs au cas de l'application complexe JPEG compilée avec la macro-option O2. Les mêmes conclusions sont obtenues pour d'autres benchmarks (annexe A.2).

III.7.1. Impact de la variation de la taille du banc de registres

Comme déjà évoqué dans la section III.3, la taille du banc de registres dépend du nombre de fenêtres qui peut varier de 2 à 32. La figure III-22 illustre les répercussions de la modification de ce paramètre.

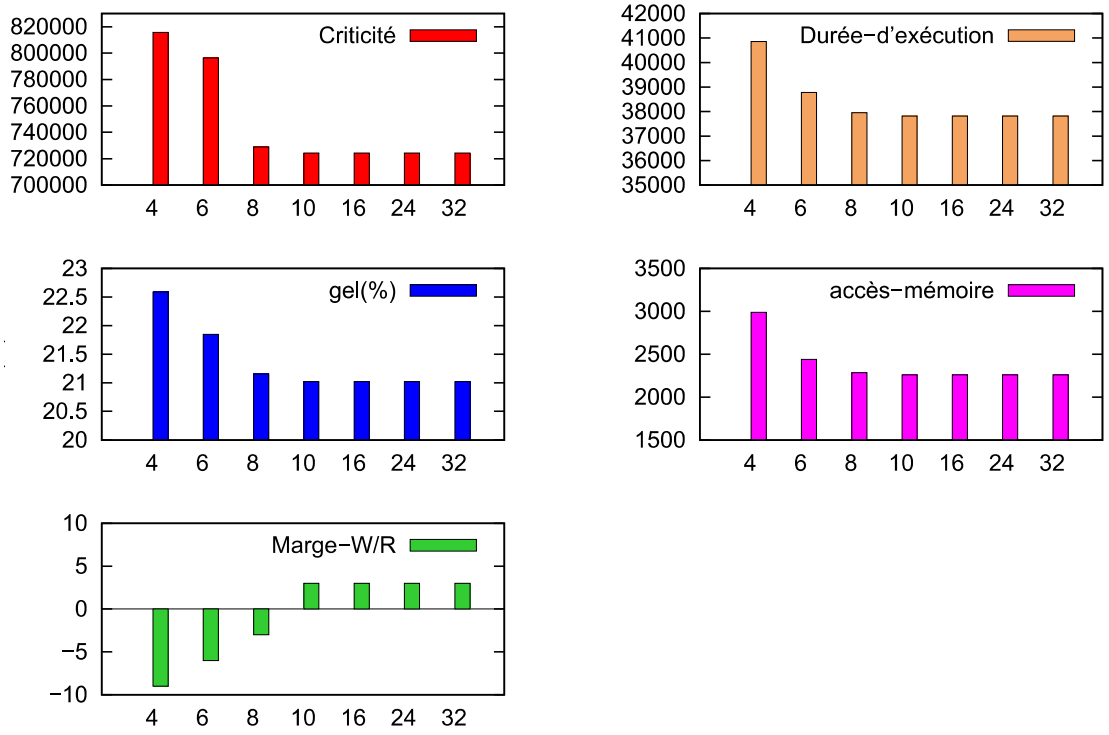


Figure III-22 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application JPEG.

Ces figures révèlent que la variation de la taille du banc de registres a des effets, d'une part, sur les performances et d'autre part sur la robustesse. En effet, l'augmentation du nombre de fenêtres diminue le nombre d'interactions avec la mémoire cache en réduisant le nombre d'écritures et de lectures. Cette diminution est due à un stockage de plus de données dans les registres. En conséquence, le nombre de cycles d'exécution et le pourcentage de gel subissent une réduction, ce qui traduit une amélioration des performances de l'application. En ce qui concerne la robustesse, nous nous apercevons que la criticité globale baisse chaque fois que le nombre de fenêtres augmente jusqu'à atteindre un seuil minimal. Ce fait peut être expliqué par la diminution du taux de blocage dans le pipeline et par l'accroissement de la marge entre le

nombre d'écritures et le nombre de lectures au niveau du banc de registres. Ceci affermit la conclusion faite dans la section III.6.2. L'évolution de la marge écritures/lectures paraît petite, mais il faut savoir qu'une seule écriture supplémentaire peut réduire tout un intervalle de cycles critiques. A partir d'un certain nombre de fenêtres, nous arrivons à un état de saturation où l'algorithme fournit les mêmes résultats en termes de performances et de criticité car l'application n'est pas en mesure d'exploiter les registres supplémentaires.

III.7.2. Impact des différentes configurations de caches

Le LEON3 dispose d'un système de mémoires configurables. Le cache de données et le cache d'instructions peuvent être paramétrés en nombre de pages ou voies (1 à 4), taille de page (1 à 256 KB), nombre de mots par ligne (4 ou 8) et politiques d'écriture et de remplacement des lignes. L'accès aux caches peut être soit à correspondance directe (cas d'une seule voie) soit à correspondance associative par sous-ensemble (cas de plusieurs voies). La figure III-23 montre les effets du type d'accès et de la taille des deux caches (instructions et données) sur la robustesse et les performances de l'application exécutée. Lors de ces expérimentations, le nombre de fenêtres dans le banc de registres est fixé à 8 (valeur par défaut), une ligne de cache peut contenir 32 octets (8 mots de 32 bits) et les politiques d'écriture et de remplacement utilisées sont respectivement l'écriture immédiate (*write-through*) et LRU (*Least Recently Used*).

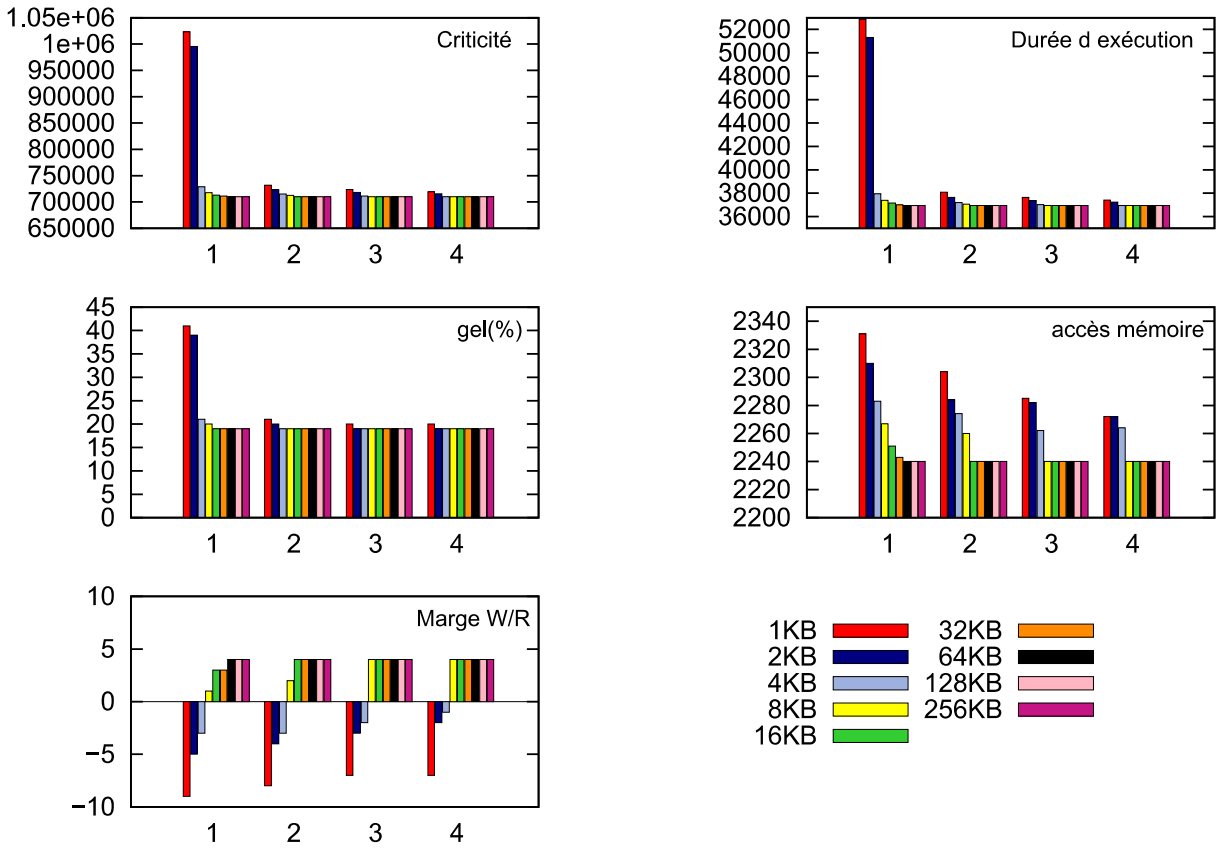


Figure III-23 : Effets de la variation du nombre de sous-ensembles et de la taille des caches.

En comparant les deux types d'accès pour une même taille de page, il s'avère que la correspondance associative par sous-ensemble offre un gain en performances sensible, qui s'interprète par une diminution de la durée d'exécution, du nombre de lectures/écritures mémoires et du pourcentage de gel dans le pipeline. Une baisse de criticité est également observée par rapport à l'accès à correspondance directe, ce qui reflète une optimisation de la robustesse. Notons que ces améliorations apparaissent lorsque le degré d'associativité des caches (nombre de voies) augmente, avec à nouveau un effet de saturation. Dans tous les cas, la réduction de la criticité globale est accompagnée par une augmentation de la marge écritures/lectures dans le banc de registres.

Les mêmes effets sont produits en augmentant la taille de page pour un mode d'accès donné.

III.8. Conclusion

Ce chapitre a décrit une nouvelle approche d'évaluation de robustesse destinée aux circuits de type microprocesseur. Cette approche se base sur un algorithme qui modélise l'architecture interne du pipeline et permet de faire une analyse dynamique du comportement des différents registres pour évaluer les criticités relatives. L'algorithme repose à son tour sur une trace d'exécution unique obtenue lors de la simulation du système. Une telle trace est normalement disponible après la validation fonctionnelle du système donc n'induit pas de travail supplémentaire pour le concepteur, contrairement à la mise en œuvre de campagnes d'injection. Le temps requis par l'approche proposée est donc en fait plus petit que les temps présentés dans ce chapitre, qui tiennent compte du temps de simulation.

L'approche a été développée pour étudier la robustesse du processeur LEON3 exécutant des applications variées. L'injection de fautes nous a permis d'obtenir des résultats de référence pour la criticité des différents registres du pipeline entier du processeur. Pour plusieurs applications, la comparaison de ces résultats avec ceux fournis par l'algorithme de prédiction a bien montré une cohérence avec une marge d'erreur acceptable ce qui est suffisant pour valider l'algorithme. L'approche proposée est beaucoup plus avantageuse que l'injection de fautes en termes de rapidité de calcul. De plus, contrairement à l'injection par émulation qui nécessite la synthèse logique et l'implémentation sur une carte de développement, l'algorithme développé n'a besoin que de la simulation fonctionnelle pour évaluer la criticité. Ceci donne la possibilité au développeur de comparer facilement et rapidement la criticité globale de son application, ou le nombre de registres à protéger, en fonction de différents choix de codage. Par rapport à des évaluations de criticité faites pendant la phase de compilation, l'approche se caractérise par la prise en compte des différentes spécificités de la microarchitecture. Cela a mis en évidence l'impact des mécanismes d'optimisation du pipeline sur la criticité réelle des registres.

Les options d'optimisation lors de la compilation ont aussi un impact sur la sûreté de fonctionnement. L'algorithme de prédiction a montré que la criticité globale d'une application varie considérablement lorsque le même code source est compilé avec différentes optimisations du compilateur. Il s'est avéré ensuite qu'éviter les optimisations est globalement le meilleur

choix en ce qui concerne la robustesse, ce qui s'accorde avec les conclusions présentées dans l'état de l'art. La modification des caractéristiques de certains blocs matériels, essentiels pour l'exécution d'une application, a également un impact fort sur la robustesse. Dans ce sens, une étude a été menée sur les effets issus de la variation des configurations du banc de registres et des mémoires caches. La rapidité des analyses permet d'obtenir une vision très complète de l'effet des choix architecturaux sur la criticité du système exécutant un logiciel donné, ce qui ne serait pas réalisable sur la base de campagnes d'injection de fautes.

L'approche présentée dans ce chapitre suppose l'utilisation d'un système à base de microprocesseur, et est surtout intéressante pour un processeur pipeline. Elle nécessite aussi un développement conséquent en cas de changement de processeur puisque les nouvelles caractéristiques micro-architecturales doivent être modélisées, pour l'ensemble du nouveau jeu d'instructions. Le chapitre suivant propose une méthode plus générique pouvant analyser la robustesse de tout circuit numérique synchrone sans hypothèse d'architecture.

Chapitre IV : Méthode d'analyse de durées de vie dans des circuits numériques

IV.1. Introduction

Dans ce chapitre nous allons présenter une nouvelle méthode d'analyse de durées de vie permettant d'estimer rapidement et avec une bonne précision la criticité des différents points de mémorisation à l'intérieur d'un circuit intégré numérique exécutant une application donnée. Les blocs mémoire ne sont pas explicitement considérés dans l'approche, car ils sont le plus souvent protégés par des approches efficaces comme des codes détecteur ou correcteur d'erreurs. L'accent est donc mis sur la sensibilité des bascules présentes dans la logique dite "aléatoire", beaucoup plus difficiles à protéger par des approches systématiques. L'objectif ici n'est pas d'avoir les valeurs exactes de criticité, comme dans le cas d'une analyse par injection de fautes exhaustive, mais d'obtenir rapidement et sans déploiement d'outils complexes un ordre de grandeur assez précis du degré de vulnérabilité d'un circuit aux fautes transitoires. Ceci doit offrir aussi au concepteur la possibilité d'identifier les éléments les plus critiques à protéger selon le degré de protection visé et, par conséquent, d'améliorer la robustesse de son circuit très tôt dans le flot de conception, c'est-à-dire avant la synthèse. Notons que l'approche est destinée à évaluer une sensibilité intrinsèque, avant application de méthodes de protection ; nous reviendrons sur ce point dans les perspectives de nos travaux.

Cette nouvelle méthode d'analyse doit répondre à trois critères principaux:

1- Généralité : la méthode proposée doit être assez générique pour qu'elle soit applicable sur un grand nombre de circuits,

2- Performances : l'analyse de criticité doit être rapide mais aussi efficace dans le sens où elle garde une bonne cohérence avec les résultats d'injections,

3- Limitation des coûts : les coûts de mise en œuvre doivent être bas, afin de garder un avantage net par rapport à la technique d'injection de fautes. Ceci inclue les compétences spécifiques requises pour pouvoir mettre en œuvre l'approche.

Nous débutons le chapitre par une présentation générale du flot d'analyse proposé. Les parties IV.3, IV.4 et IV.5 donnent une description détaillée des principales étapes de traitement de ce flot. En vue d'évaluer notre méthode, nous nous intéresserons dans la partie IV.6 à appliquer l'outil de calcul de criticité sur plusieurs circuits de taille significative afin de comparer les résultats obtenus avec ceux issus de campagnes d'injection de fautes. La dernière partie de ce chapitre portera sur une comparaison avec l'approche décrite dans le chapitre 3 pour le cas du processeur LEON3.

IV.2. Flot d'analyse de durées de vie

IV.2.1. Présentation générale

Un circuit numérique (blocs mémoires mis à part) est constitué de logique combinatoire et d'un ensemble de bascules et/ou verrous mémorisant chacun un bit d'information. Dans le contexte des circuits modernes, obtenus par synthèse à partir de descriptions comportementales, la très grande majorité des mémorisations est réalisée dans des bascules ; nous nous limiterons donc essentiellement à ce cas dans la suite de ce chapitre.

Le rôle d'une bascule est de mémoriser une information le temps nécessaire à son traitement ou à son utilisation dans des traitements (une ou plusieurs période(s) d'horloge). Les valeurs mémorisées définissent l'état logique du circuit à un instant donné. Si l'une de ces valeurs subit une faute transitoire, le circuit passe dans un état logique erroné pouvant conduire à un comportement inacceptable si la valeur modifiée a un impact sur les traitements exécutés. L'analyse des bits pouvant conduire à chaque cycle à un tel comportement donne donc une image de la criticité des différentes bascules et permet de prioriser la protection de celles qui sont les plus sensibles. Une image de la criticité par cycle d'exécution est obtenus de même en considérant l'ensemble des bascules à chaque cycle, et la criticité (ou sensibilité) globale est obtenue en considérant toutes les bascules, sur l'ensemble des cycles d'exécution de l'application sélectionnée.

La durée de vie d'une bascule est égale au nombre de cycles où une faute survenant sur cette dernière aurait des effets non désirables sur le résultat attendu à la fin de l'exécution. Le pourcentage de criticité relatif à cette bascule est alors calculé, pour une exécution donnée, en faisant le rapport entre cette durée et le nombre total de cycles. Le flot d'analyse que nous proposons a pour but d'identifier l'ensemble des cycles critiques pour chaque bascule. Le principe de ce flot est illustré dans la figure IV-1.

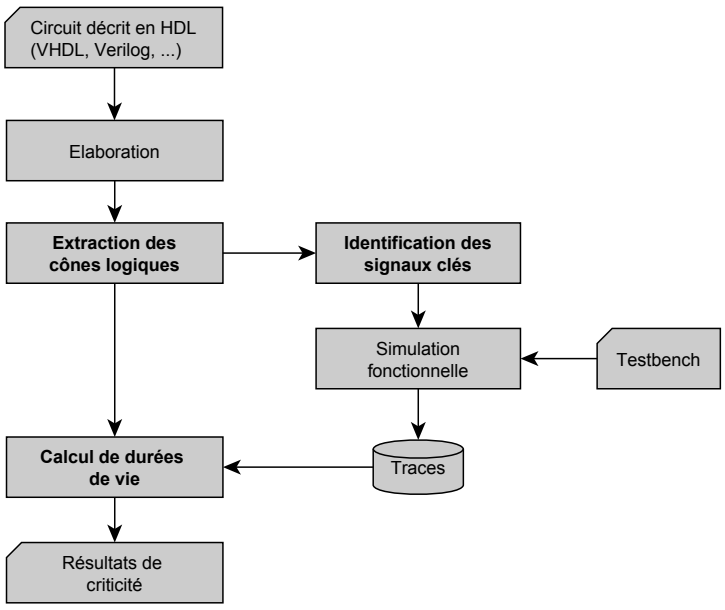


Figure IV-1 : Vue globale du flot d'analyse de durées de vie.

Une description matérielle de haut niveau du circuit cible doit être fournie (classiquement une description au niveau RTL ou comportementale). Un fichier de stimuli (*testbench*) est aussi nécessaire pour pouvoir simuler le circuit pendant l'exécution de l'application ciblée.

D'abord, l'élaboration est lancée dans le but d'avoir une vue structurelle du circuit (*netlist*), sans hiérarchie et en fonction de composants de base. Ces composants sont génériques, sans lien avec une technologie particulière, et représentent donc uniquement le comportement logique attendu. La *netlist* générique est obtenue très rapidement, par interprétation de la description comportementale ; il ne s'agit pas d'une synthèse, au sens où aucune contrainte de technologie ou de performances (fréquence d'horloge, consommation, …) n'est prise en compte à ce stade et il n'y a pas d'optimisation réalisée. Ceci implique aussi que les analyses réalisées sont très génériques car n'utilisent que des contraintes comportementales provenant de la spécification fonctionnelle.

L'élaboration peut être réalisée avec différents types d'outils disponibles dans le flot de conception classique (simulateurs ou outils de synthèse). Il suffit d'enregistrer le résultat de la première étape de travail de ces outils, sans exécuter une simulation ou une synthèse. Tout concepteur de circuits numériques a un tel outil disponible. Dans notre cas, nous avons surtout utilisé le logiciel Modelsim de *Mentor Graphics*, qui supporte plusieurs langages de description, dont *Verilog*, *SystemVerilog*, *VHDL* et *SystemC*. Un autre choix d'outil ne changerait pas la méthodologie, mais juste le format de fichier à prendre en compte pour l'automatisation du flot.

Ensuite, l'étape "Extraction des cônes logiques" s'occupe de traiter cette *netlist* pour extraire des informations sur l'architecture interne du circuit comme la liste des bascules et leurs cônes logiques (nous reviendrons dans la section IV.3 sur la définition d'un cône logique). A partir de ces informations, l'étape "Identification des signaux clés est responsable d'identifier les signaux considérés comme les plus importants, car ils contrôlent le chargement au niveau de chaque bascule. Puis, à l'aide du *testbench*, une simulation fonctionnelle classique permet d'enregistrer les traces relatives aux signaux précédemment identifiés. Enfin, en connaissant les caractéristiques du circuit, l'étape "Calcul de durées de vie" exploite ces traces de simulation pour calculer la durée de vie de chaque bascule et conclure par la suite sur sa criticité. La même étape peut donner les informations de criticité globale, ou par cycle.

L'utilisateur fixe dès le début les cycles d'observation du circuit, c'est-à-dire les cycles auxquels les sorties du circuit doivent être conformes au résultat attendu. Autrement dit, ce sont les cycles où l'utilisateur exploite les sorties primaires du circuit.

Il est aussi possible de fixer la liste des sorties qui doivent être prises en compte pour évaluer la criticité.

IV.2.2. Hypothèses

Afin de mettre en place notre approche d'évaluation de durées de vie, nous avons tout d'abord fixé un cadre de travail. Les hypothèses que nous avons formulées sont les suivantes :

a) Nous nous plaçons dans le cas de circuits synchrones ayant une horloge unique. Cela signifie que nous ne prenons pas en compte, à ce stade, la génération d'horloges internes dérivées (par exemple par le biais de compteurs) ou la technique de validation d'horloge (appelée en anglais *clock gating*) qui est une méthode de réduction de la consommation dynamique d'un circuit consistant à couper le signal d'horloge d'une partie du circuit lorsque celle-ci est inactive. Les circuits multi-horloges ne sont de même pris en compte que par bloc ayant une même horloge (appelés domaines d'horloge) et les interfaces entre ces blocs ne sont pas analysées.
b) Toutes les bascules sont des bascules D synchronisées sur le même front d'horloge (montant ou descendant). Il faut noter que la plupart des circuits obtenus par synthèse logique ont cette caractéristique.
c) Les traces de simulation doivent contenir les états des signaux de contrôle utiles à chaque front actif de l'horloge, autrement dit, à chaque instant où toutes les bascules du circuit sont chargées. Si une entrée asynchrone d'une bascule (cas des *set/reset* asynchrones) subit un changement d'état, nous supposons que ce changement dure au moins une période d'horloge (ou est au moins recouvrant par rapport au front actif) afin que ceci soit enregistré dans les traces obtenues après simulation. Il est à noter que ces signaux ont un impact fort sur la criticité puisqu'ils sont capables de modifier le contenu de la bascule à tout moment sans attendre le front d'horloge.
d) Le flot d'analyse que nous proposons dans cette thèse ne tient pas compte des situations de masquage logique qui peuvent se produire dans la logique combinatoire située entre les bascules, sauf dans un cas particulier qui sera précisé dans les sections suivantes. Pour une porte logique, la valeur d'une entrée peut figer la valeur de sortie de la porte, indépendamment de la valeur des autres entrées. Par exemple, si l'une des entrées d'une porte ET à 2 entrées a une valeur '0' logique, la sortie sera 0, quelle que soit la valeur de l'autre entrée. Ainsi, lorsqu'une valeur erronée est propagée jusqu'à une porte dont la valeur de sortie est figée par une autre entrée, la propagation de l'erreur s'arrête, et il y a un masquage logique de la faute. Sur la figure IV-2, nous trouvons une représentation basique de ce phénomène. Ce type de masquage n'est pas pris en compte dans notre cas afin de simplifier, et donc accélérer, l'analyse réalisée tout en limitant le nombre de signaux à enregistrer lors des simulations. Par ailleurs, une prise en compte efficace de tous les masquages logiques ne serait possible qu'après une synthèse définitive sur la technologie cible ; notre analyse a pour objectif d'être réalisable beaucoup plus tôt, sans tenir compte d'une technologie donnée ou d'un ensemble précis de contraintes de synthèse.

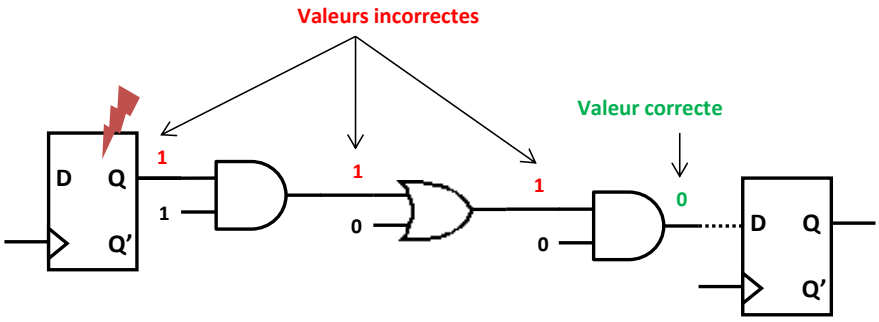


Figure IV-2 : Masquage logique d'une faute transitoire.

IV.3. Extraction des cônes logiques

IV.3.1. Cône logique

A un niveau bas d'abstraction, un circuit numérique est vu comme l'interconnexion de bascules et de portes logiques. Le cône logique d'une bascule donnée est l'ensemble des portes logiques qui définissent la valeur de son entrée. Son sommet est l'entrée de la bascule elle-même, et sa base représente généralement les bascules et les entrées primaires qui y sont reliées à travers le réseau combinatoire. Ceci est illustré dans la figure IV-3.

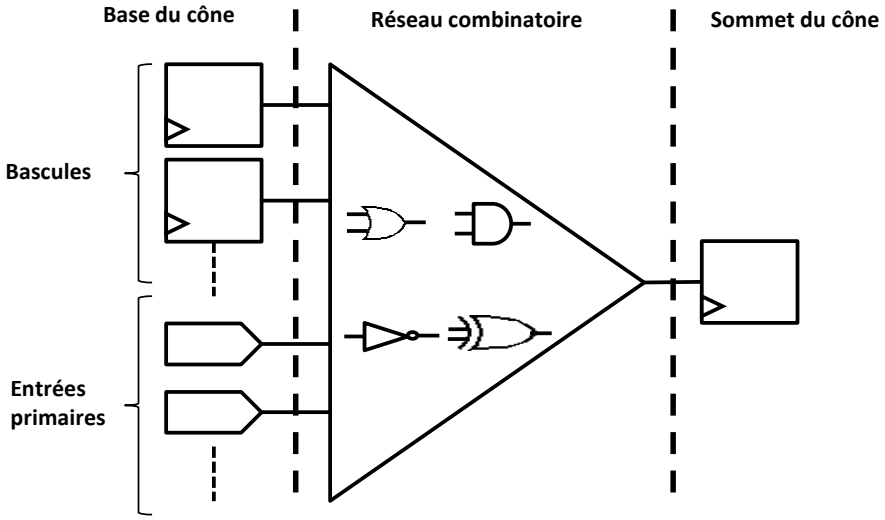


Figure IV-3 : Schéma d'un cône logique.

Certaines bascules peuvent être connectées directement ou via de la logique combinatoire uniquement sur des ports d'entrée. Dans ces cas, la bascule est dite "primaire" et son cône n'est pas rattaché à des bascules. Dans les autres cas, la bascule est dite "interne" et peut être alimentée par une ou plusieurs bascules.

Nous pouvons définir également les cônes logiques relatifs aux sorties primaires du circuit. Dans ce cas, le sommet du cône n'est plus une bascule mais un signal de sortie. En conséquence, le nombre total de cônes est égal au nombre de bascules additionné avec le nombre de signaux de sortie.

Ces cônes décrivent les dépendances fonctionnelles existant entre les différents éléments de mémorisation qui constituent le circuit. De ce fait, un circuit peut être modélisé par un ensemble de cônes logiques qui interagissent entre eux pour réaliser une fonction bien déterminée.

IV.3.2. Algorithme d'extraction

L'organigramme représenté en figure IV-4 résume les étapes suivies dans l'algorithme pour parvenir à identifier précisément les cônes logiques. Nous décrirons de façon plus détaillée chaque étape dans les paragraphes qui suivent.

Il faut noter que des environnements commerciaux existent pour faire ce type d'extraction. Dans notre cas, il a été préféré de développer nos propres procédures afin de mieux maîtriser les informations extraites, dans l'optique de faciliter les étapes suivantes du flot d'analyse de durées de vie.

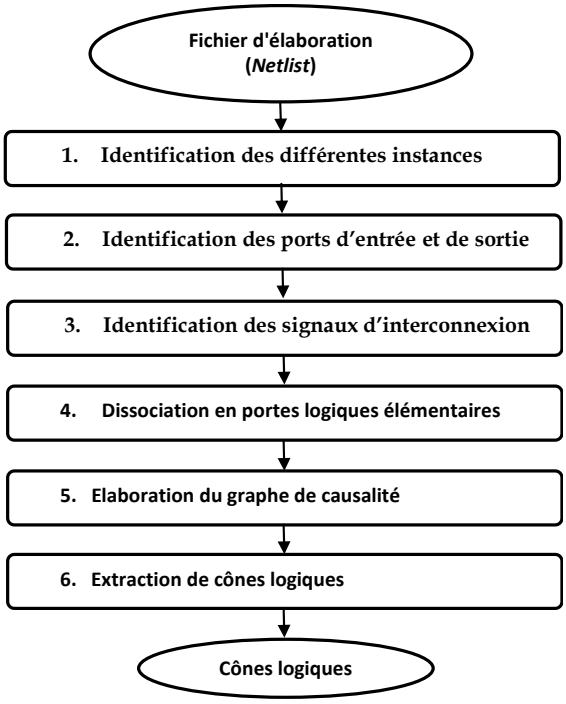


Figure IV-4 : Processus d'extraction des cônes à partir des données d'élaboration.

IV.3.2.1. Identification des instances, des ports E/S et des signaux d'interconnexion

La première étape consiste à identifier les instances (portes logiques combinatoires et séquentielles) qui ont été utilisées lors de l'élaboration du circuit. Chaque instance est caractérisée par :

- Un nom,
- Un type (and, or, mux, bascule, etc.),
- Une liste des signaux d'entrée,
- Une liste des signaux de sortie.

De la même manière que pour les instances, la deuxième étape extrait les différents ports d'entrée/sortie du circuit. De ce fait, chaque port est caractérisé par :

- Un nom,
- Une direction (entrée/sortie).
- Liste des signaux qui y sont associés.

Après ces deux étapes, tous les composants de base du circuit sont déterminés et associés chacun à un modèle. Pendant la troisième étape, l'algorithme identifie les signaux qui relient les différentes instances les unes aux autres. De plus, il fait la liaison avec les ports d'entrée/sortie du circuit. Toutes les connexions entre les éléments du circuit sont identifiées ainsi il devient possible de connaitre à quels éléments et à travers quels signaux chaque instance est liée.

IV.3.2.2. Etape 4 : Dissociation en portes logiques élémentaires

L'algorithme dissocie en portes logiques dites élémentaires toute instance dont la taille de la sortie est supérieure à 1, le processus d'élaboration utilisant le plus souvent une notion d'élément multi-bits, avec des bus en entrée et en sortie. Le but de cette étape est de mieux distinguer les signaux qui pourraient influencer l'état de chaque signal de sortie. Par exemple dans le cas de la figure IV-5, la sortie c1 ne dépend pas de toutes les entrées mais seulement des entrées a1, b1 et d. De même, c2 ne dépend que de a2, b2 et d. Ceci est particulièrement important pour affiner l'analyse de chemins de propagation non réguliers (c'est-à-dire par exemple le cas où, à l'intérieur d'un circuit, un bit d'indice 0 sur un bus de sortie est connecté à un bit d'indice 4 sur un bus d'entrée).

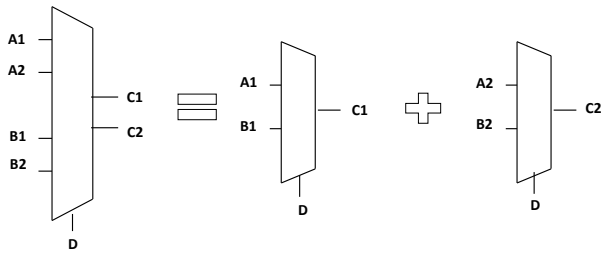


Figure IV-5 : Dissociation d'une instance de nature multiplexeur 2 vers 1.

IV.3.2.3. Etape 5 : Elaboration du graphe de causalité

Le graphe de causalité modélise les différentes dépendances existant entre les éléments constituant le circuit à analyser. Autrement dit, il définit les relations directes de cause à effet entre les signaux des différentes instances et des ports d'E/S. Les nœuds de ce graphe représentent les composants élémentaires (portes combinatoires/séquentielles et les ports d'E/S). Les arcs, quant à eux, représentent une relation de causalité directe entre deux composants. La figure IV-6 illustre le graphe de causalité d'un exemple de circuit.

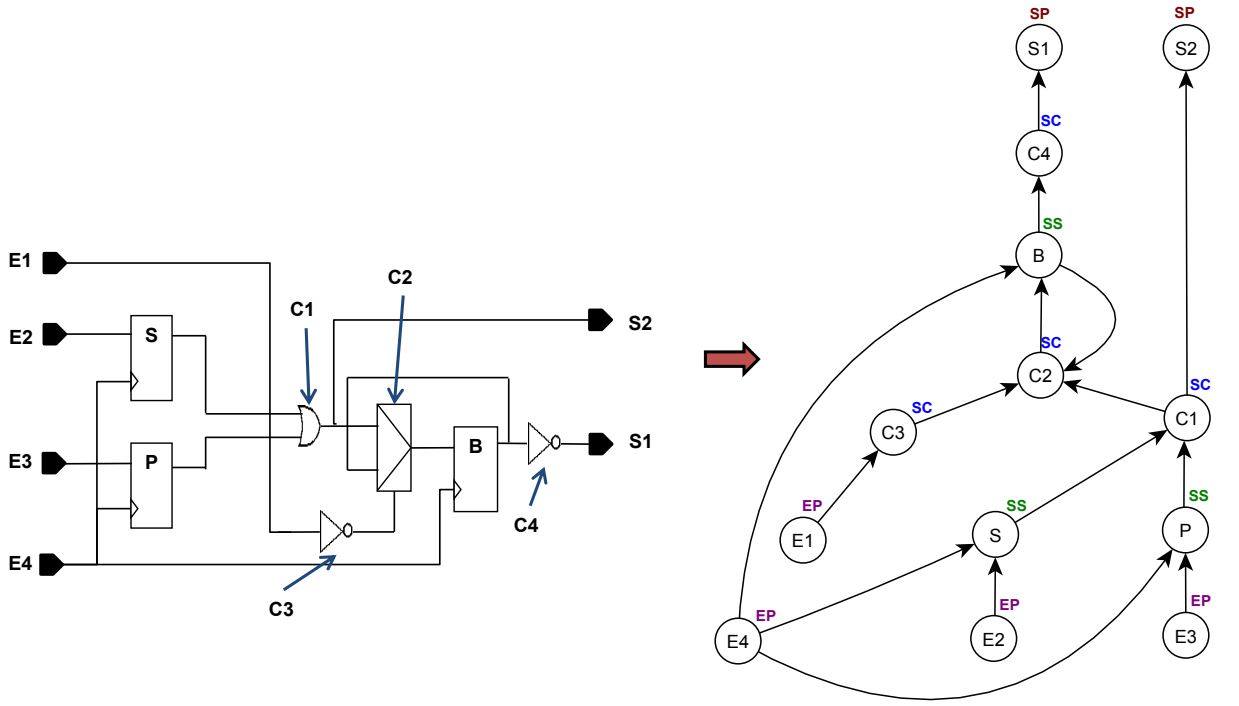


Figure IV-6 : Exemple de graphe de causalité.

Ce graphe stocke toutes les informations nécessaires pour l'extraction des cônes. Il correspond à un ensemble de vecteurs qui représentent chacun un signal de sortie (d'une porte logique donnée ou une sortie primaire) et la liste des signaux de causalité, c'est-à-dire ceux qui pourraient être la cause d'un changement d'état du signal de sortie. Chaque élément du graphe de causalité est caractérisé par :

- Un nom
- Une nature
- Une origine

Comme mentionné précédemment, le premier élément de chaque vecteur est un signal de sortie dont l'origine est l'élément qui le génère. La nature de ce signal peut être alors :

- Soit une sortie primaire **(SP)** : si le signal correspond à une sortie primaire du circuit.
- Soit une sortie combinatoire **(SC)** : si le signal correspond à une sortie d'une instance combinatoire à l'intérieur du circuit. L'origine du signal sera par conséquent le nom de l'instance combinatoire.
- Soit une sortie séquentielle **(SS)** : si le signal correspond à une sortie d'une instance séquentielle. Alors, l'origine du signal sera le nom de l'instance séquentielle.

Pour les signaux de causalité, l'algorithme fait recours à une autre procédure de classification. En effet, la nature de chaque signal peut être soit une sortie combinatoire (SC) ou séquentielle (SS) soit une entrée primaire **(EP)**. Dans ce dernier cas, l'origine du signal sera affectée au nom du port d'entrée relatif.

Afin de mieux comprendre cette notion de causalité, prenons le cas de la figure IV-6. Le graphe présenté montre que les signaux de sortie des portes combinatoires C1 et C3 et la sortie de bascule B influent sur l'état du signal généré par la porte C2. Ces trois signaux représentent donc les signaux de causalité de C2.

IV.3.2.4. Etape 6 : Extraction des cônes logiques

Maintenant que le graphe de causalité est déterminé, l'algorithme passe à la dernière étape qui consiste à extraire les cônes logiques. Dans le but de faciliter la compréhension, nous allons focaliser notre étude sur une bascule que l'on va nommer "B". Ensuite, les différentes constatations seront applicables sur le reste des bascules. Pour déterminer le cône logique de B, l'algorithme se base entièrement sur le graphe de causalité élaboré pendant l'étape 5. D'abord, il commence par identifier le signal d'entrée de donnée de B. Si le signal identifié correspond à une entrée primaire alors B sera considéré comme une bascule primaire. Si ce n'est pas le cas, alors ce signal sera projeté dans le graphe de causalité car il pourrait être soit la sortie d'une instance combinatoire, soit la sortie d'une instance séquentielle.

- S'il s'agit d'une sortie d'instance séquentielle (c'est-à-dire une sortie de bascule), l'algorithme ajoute cette bascule au cône logique de B.
- Dans le cas contraire, la porte combinatoire est rajoutée au cône de B. Ensuite, l'algorithme examine la liste des signaux de causalité. Dans cette liste, tout élément de nature entrée primaire ou bien sortie séquentielle sera ajouté au cône de B. Par contre, tout signal de nature sortie combinatoire sera projeté de nouveau dans le graphe de causalité afin d'examiner ses signaux de causalité. Ainsi, l'algorithme refait les mêmes traitements plusieurs fois jusqu'à ce qu'il arrive à extraire toutes les bascules et entrées primaires qui alimentent B.

Puisque l'algorithme ne connait pas à l'avance le nombre de portes combinatoires qu'il va traiter, ce processus d'identification a été implémenté sous la forme d'une fonction récursive. La condition d'arrêt de cette fonction est validée quand toutes les sorties combinatoires dans le cône logique sont évaluées dans le graphe de causalité et exprimées en fonction des bascules et/ou des entrées primaires. La figure IV-7 représente un organigramme résumant la méthode de raisonnement adoptée.

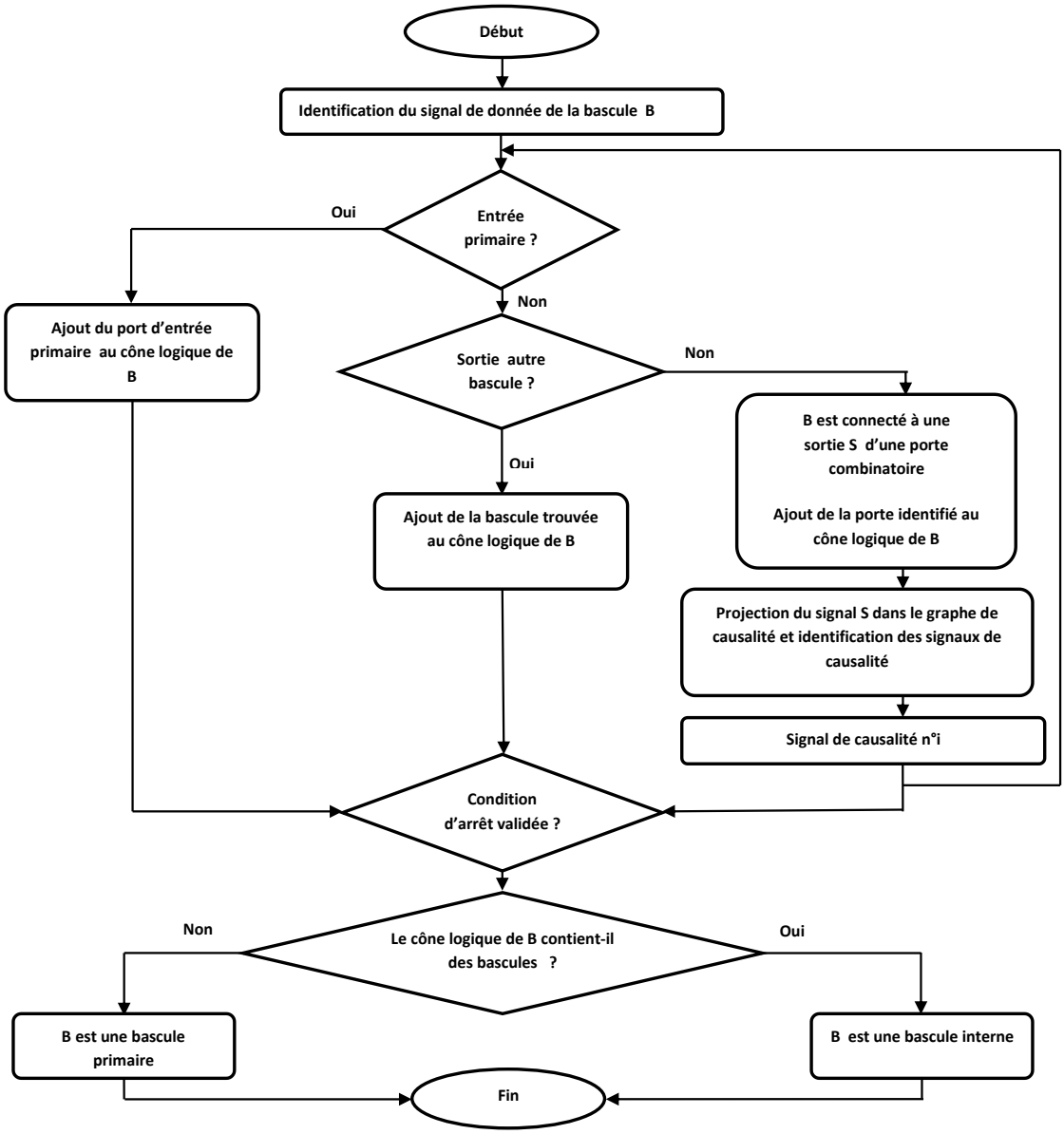


Figure IV-7 : Organigramme d'identification des cônes logiques.

En cas de bascule ayant des entrées de contrôle, deux cas se présentent pour chaque entrée. Si celle-ci est asynchrone, les mêmes démarches sont à suivre afin que les entrées primaires ou les bascules liées à cette entrée soient aussi identifiées. Le point de départ dans ce cas n'est plus l'entrée de donnée mais l'entrée de contrôle. En revanche, si elle est synchrone, elle sera modélisée lors de l'élaboration comme l'entrée de sélection d'un multiplexeur placé juste avant l'entrée de donnée de la bascule du sommet de cône (figure IV-8). L'entrée va être, en conséquence, traitée implicitement dans le processus d'identification puisqu'elle présente un signal de causalité pour l'entrée de donnée de la bascule. Ceci nous amène à déduire qu'aucun autre traitement n'est nécessaire dans le cas des entrées synchrones.

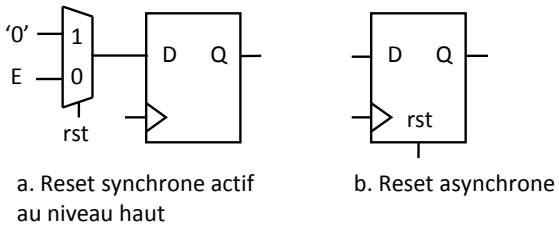


Figure IV-8 : Exemple de reset synchrone (a) et de reset asynchrone (b) actifs au niveau haut.

Pour les cônes des sorties primaires, il suffit de changer le signal de départ, c'est-à-dire, le signal d'entrée de donnée par celui définissant la sortie primaire. De ce fait, la méthode nous permet de détecter rigoureusement tous les cônes logiques qui pourraient exister dans un circuit numérique donné. L'algorithme caractérise chaque cône par l'ensemble des bascules, l'ensemble des portes combinatoires génériques après élaboration et l'ensemble des entrées primaires qui peuvent avoir un effet sur l'état de la bascule ou de la sortie qui représente le sommet du cône logique analysé. En outre, il est bien possible de faire la distinction entre bascule primaire et bascule interne. Un autre résultat de l'analyse est que toute situation de rebouclage d'information sera évidemment détectée par l'algorithme d'identification. Comme exemple, nous allons prendre le cas du circuit de la figure IV-6 et essayer de déterminer le cône logique de la bascule B en suivant la méthode proposée.

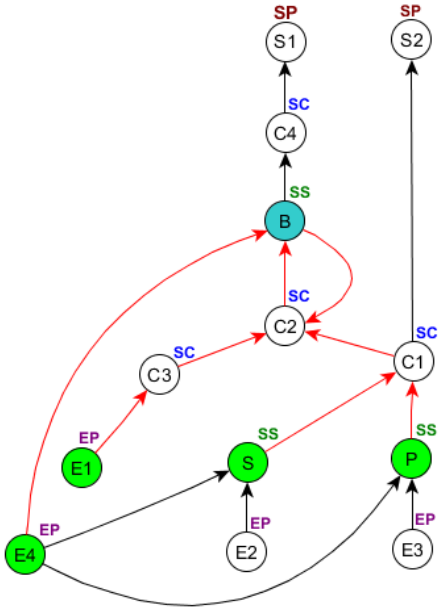


Figure IV-9 : Identification des éléments du cône logique associé à la bascule B.

D'après la figure IV-9, le cône logique de B est sous la forme suivante :

- Nom du cône : bascule **B**,
- L'ensemble des bascules : **S**, **P** et **B,**
- L'ensemble des portes combinatoires : **C1**, **C2** et **C3,**
- L'ensemble des signaux d'entrée primaires: **E1** et **E4.**

Nous pouvons remarquer que B fait partie de la liste des bascules sources de son cône. Ceci montre que l'algorithme détecte qu'il y a une dépendance entre l'entrée et la sortie de cette bascule. Ceci correspond, le plus souvent, à une validation de chargement par "validation de données", afin d'identifier les cycles où une nouvelle information utile doit être chargée dans la bascule.

IV.4. Identification des signaux clés

Une fois les différents cônes logiques identifiés, les signaux indispensables à l'analyse des durées de vie doivent être extraits. Cette étape est très critique dans le flot d'analyse parce qu'elle va identifier les signaux qu'il faudra suivre pendant la simulation fonctionnelle du circuit montrant l'exécution d'une application pour générer les traces qui conduiront au calcul des durées de vie.

IV.4.1. Notion de branche logique

En partant de la définition donnée dans la partie IV.3.1, un cône logique peut être représenté de manière informatique sous la forme d'un arbre dont la racine est le sommet du cône et les feuilles sont les bascules et les entrées primaires qui forment la base du cône. Nous appelons branche logique tout chemin reliant une feuille à la racine. Dans la plupart des cas, le chemin de liaison feuille-racine contient des nœuds correspondant à des portes combinatoires à travers lesquelles la feuille agit sur la racine. La figure IV-10 donne un exemple de représentation d'un cône à l'aide des branches logiques.

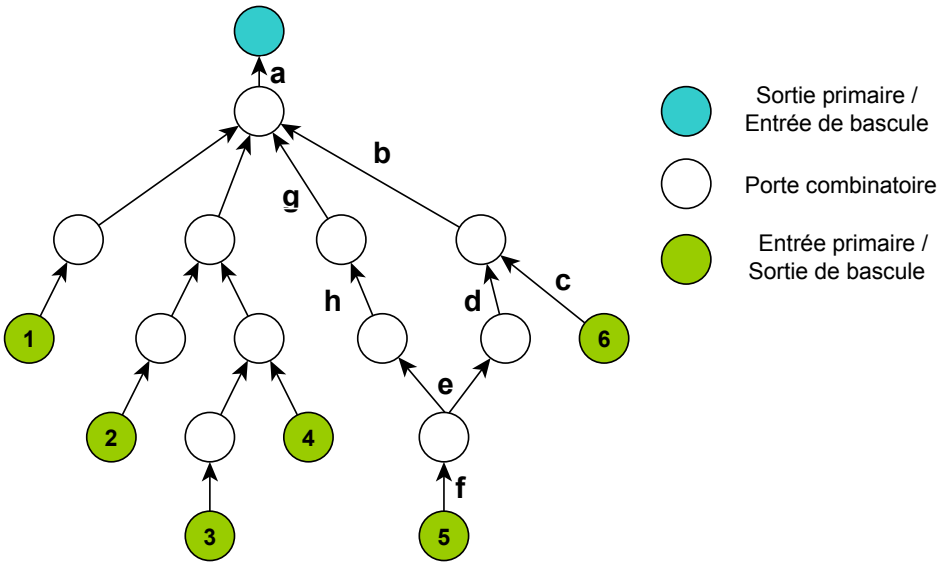


Figure IV-10 : Représentation d'un cône en fonction de branches logiques.

La branche définie par les signaux (c, b, a) dans la figure IV-10 lie la feuille numéro 6 à la racine. Autrement dit, le signal c généré par cette feuille peut influencer le signal b, ce dernier peut à son tour influer sur la valeur du signal a qui sera ensuite chargée dans la racine (port de sortie ou entrée de bascule). Pour atteindre la racine, il se peut que plusieurs chemins puissent être suivis. C'est le cas de la feuille 5 où les deux branches (f, e, d, b, a) et (f, e, h, g, a) mènent à la racine. Cela vient du fait que le signal e alimente deux portes combinatoires à la fois.

IV.4.2. Validation de branche logique et identification des signaux clés

A un cycle d'exécution donné, une feuille peut avoir ou ne pas avoir un effet sur la racine. Ceci dépend des valeurs des feuilles et de la nature des éléments combinatoires qui lient les feuilles et la racine. En d'autres termes, pour qu'une feuille ait un effet sur la racine il faut que certaines conditions soient remplies. Toute branche associée à cette feuille et qui respecte ces conditions est dite "branche valide". Toutefois, comme indiqué précédemment, nous ne cherchons pas à analyser tous les cas de masquages logiques afin de réduire la complexité et la durée de nos évaluations. En conséquence, seules les portes génériques de type multiplexeur sont prises en compte dans l'identification des signaux clés, car les multiplexeurs permettent d'éliminer de nombreuses possibilités de propagation à partir de peu d'informations. En particulier, ils permettent d'identifier en général les cas de validations de données. Ils permettent aussi à chaque cycle d'éliminer certains chemins (le signal de commande ne pouvant sélectionner qu'une entrée sur deux), ce qui n'est pas le cas des autres portes logiques, beaucoup plus dépendantes de l'état de l'ensemble de leurs entrées.

Comme exemple, nous analysons le cône de la bascule B du circuit de la figure IV-11.

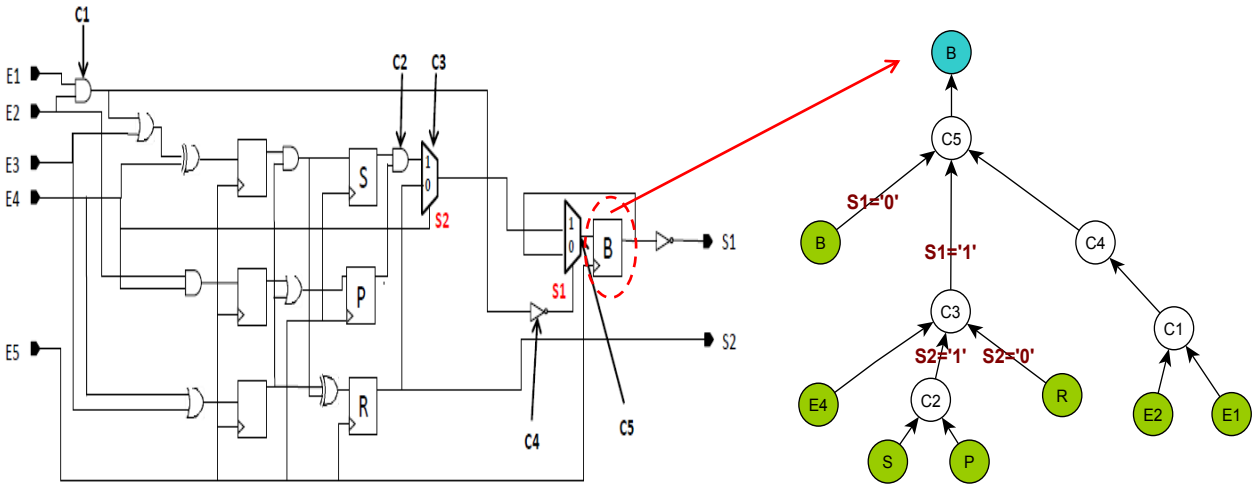


Figure IV-11 : Analyse des branches logiques du cône B.

Nous pouvons remarquer que la validation des branches provenant de B, E4, S, P, et R dépend, à chaque cycle, des états des signaux S1 et S2. Ces signaux représentent, par conséquent, les signaux clés du cône de B parce qu'ils contrôlent son chargement et décident des signaux qui contribuent à la définition de sa valeur à un cycle donné. A titre d'exemple, si au front actif de l'horloge, S1 vaut 0 alors les signaux E4, S, P et R n'interviendront pas dans le calcul de la nouvelle valeur de B.

Nous donnons dans la suite les conditions de validation de chaque branche :

- La branche B-B est valide si, et seulement si, S1=0.
- La branche B-E4 est valide si, et seulement si, S1=1.
- Les branches B-S et B-P sont valides si, et seulement si, S1=1 et S2=1.
- La branche B-R est valide si, et seulement si, S1=1 et S2=0.

En ce qui concerne les branches E1 et E2, il n y a pas de condition de validation. Nous supposons que ces feuilles ont toujours un impact potentiel sur l'entrée de donnée de la bascule B même si leur valeur pourrait être masquée par un phénomène de masquage logique.

Dans un premier temps, l'analyse d'une branche logique vise donc à repérer les nœuds du graphe (instances combinatoires) correspondant aux multiplexeurs et à établir la condition de validation de chaque branche.

En cas de bascule avec des entrées asynchrones, vu leur importance, ces entrées sont ajoutées automatiquement à la liste des signaux clés.

L'identification des signaux clés est l'étape la plus longue et la plus répétitive dans le flot proposé, il faut donc l'automatiser entièrement pour fournir les résultats sans intervention de l'utilisateur. Pour ce faire, un algorithme a été conçu. Celui-ci reçoit en entrée les données sur les cônes, déduit la condition de validation de chaque branche et délivre en sortie la liste des signaux à observer durant la simulation fonctionnelle du circuit. Les étapes de l'algorithme sont détaillées dans la figure IV-12.

Algorithme d'identification des signaux Clés

```
Entrée : Cônes logiques
Résultat : Liste des signaux clés
Pour i allant de 1 à nbr_cones (pas=1) faire
    Pour j allant de 1 à nbr_branches_cone_i (pas=1) faire
        Condition_validation_branche_j
        Si (Branche relative à une entrée asynchrone) alors
            + Déterminer à quel niveau cette entrée est active
            + Sauvegarde de l'information dans Condition_validation_branche_j
            Si (l'entrée ne figure pas dans la liste des signaux clés) alors
                + Ajout de l'entrée à la liste
            + Passer à la branche suivante
        Pour k allant de 1 à nbr_éléments_branche_j (pas=1) faire
            Si (Elément[k] est une instance d'aiguillage) alors
                + Identification des signaux de sélection
                + Déterminer à quelles valeurs ces signaux assurent le transfert de
                l'état du signal de sortie relatif à l'Elément [k-1]
                + Sauvegarde de l'information dans Condition_validation_branche_j
                Pour p allant de 1 à nbr_signaux_selection (pas=1) faire
                    Si (signal[p] ne figure pas dans la liste des signaux clés) alors
                        + Ajout de signal[p] à la liste
```

Figure IV-12 : Algorithme d'identification des signaux clés.

Essayons maintenant d'appliquer cet algorithme sur l'exemple de branche donné en figure IV-13.

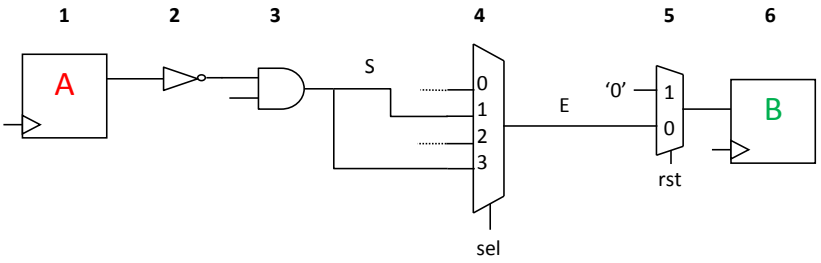


Figure IV-13 : Branche logique liant la bascule A (feuille) à la bascule B (racine).

Le quatrième élément de la branche est un multiplexeur 4 vers 1. Pour que cette instance fasse propager le signal S (sortie du troisième élément) qui dépend de l'état de la bascule A, il faut que le signal sel soit égal à 1 ou à 3. Ensuite, afin que la bascule B soit chargée par la valeur de E, il est nécessaire que le signal rst soit à l'état 0. Les signaux rst et sel sont donc ajoutés à la liste des signaux clés. Ainsi, la condition de validation de la branche est la suivante :

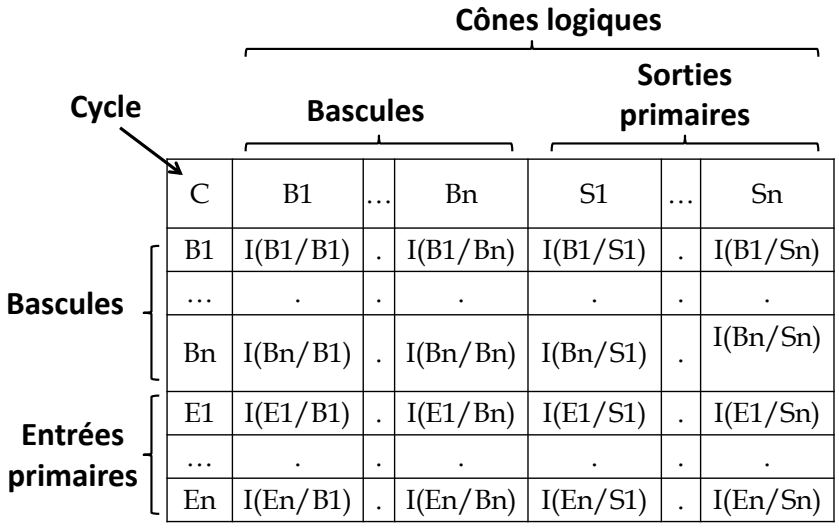
- (sel=1 ou sel=3) et rst=0

IV.5. Calcul de durées de vie

La dernière phase de l'analyse consiste à déterminer la durée de vie des différentes bascules en vue d'évaluer leur criticité selon l'application (et/ou le scenario) choisi. Notre flot d'analyse doit donc être en mesure d'estimer, pour chaque bascule, le nombre de cycles pendant lesquels l'apparition d'une faute transitoire risque de provoquer une défaillance du circuit. Avant de décrire l'algorithme adopté pour le calcul de durées de vie, nous présentons tout d'abord la notion de la matrice d'impacts.

IV.5.1. Matrice d'impacts

La matrice d'impacts (figure IV-14) est une structure de données permettant de stocker, à chaque cycle d'exécution, l'impact potentiel de chaque bascule et de chaque entrée primaire sur l'ensemble des cônes logiques du circuit.



C	B1	…	Bn	S1	…	Sn
B1	I(B1/B1)	.	I(B1/Bn)	I(B1/S1)	.	I(B1/Sn)
…	.	.	.	.	.	.
Bn	I(Bn/B1)	.	I(Bn/Bn)	I(Bn/S1)	.	I(Bn/Sn)
E1	I(E1/B1)	.	I(E1/Bn)	I(E1/S1)	.	I(E1/Sn)
…	.	.	.	.	.	.
En	I(En/B1)	.	I(En/Bn)	I(En/S1)	.	I(En/Sn)

Figure IV-14 : Matrice d'impacts.

Soit I la fonction qui définit cet impact. Par exemple, la case représentée par **I**(B1/Bn) exprime l'impact de la bascule B1 sur le cône logique de la bascule Bn au cycle C. Nous notons alors :

$$\begin{cases} \mathbf{I}(B1/Bn)=1 \text{ si la branche logique (B1, Bn) est valide au cycle C} \\ \mathbf{I}(B1/Bn)=0 \text{ sinon} \end{cases}$$

Au cas où B1 ne fait pas partie de la base du cône de Bn alors **I**(B1/Bn) est toujours égal à 0. Dans le cas inverse, le résultat va dépendre de la branche logique (B1, Bn). Si les conditions de validation de cette branche ne sont pas remplies alors I(B1/Bn) vaut 0. Par contre, si (B1, Bn) est valide alors B1 est considérée comme potentiellement vivante par rapport à Bn et **I**(B1/Bn) est égal à 1. Les signaux clés jouent donc un rôle de premier plan dans la définition de la matrice d'impacts parce que c'est l'état de ces signaux qui décide de valider ou d'invalider les branches logiques à chaque cycle d'horloge (section IV.4.2).

IV.5.2. Algorithme de calcul de durées de vie

L'algorithme que nous proposons ici se base essentiellement sur la matrice d'impacts présentée précédemment : les traces des signaux clés obtenues après la simulation permettent à l'algorithme de déterminer les bascules dites potentiellement vivantes à chaque cycle. L'utilisateur fixe dès le début les cycles d'observation du circuit, c'est-à-dire les cycles auxquels les sorties du circuit doivent être conformes au résultat attendu. Autrement dit, ce sont les cycles où l'utilisateur exploite les sorties primaires du circuit.

Pour chaque cycle d'observation, l'algorithme analyse l'historique d'exécution afin de connaitre pendant quels cycles et à quels endroits (bascules/entrées primaires) une faute transitoire aurait pu perturber le résultat obtenu à l'instant d'observation. Nous détaillons dans la figure IV-15 l'algorithme de calcul.

Algorithme de calcul de durées de vie

Entrée : liste des cycles d'observation, Cônes logiques, Traces de simulation
Résultat *: la durée de vie de chaque bascule et la criticité correspondante*
Liste des cônes (colonnes) à évaluer dans la matrice d'impacts : ***LISTC***
Bascules potentiellement vivantes par cycle : ***BPVC***
Pour chaque cycle d'observation (C) **faire**

+ Initialisation de la matrice d'impacts à zéro
+ Initialisation de ***LISTC*** aux colonnes des sorties primaires observées
1+ Déterminer, pour chaque élément de ***LISTC***, l'ensemble des valeurs de la fonction **I**
(Utilisation des traces de simulation pour la validation des branches logiques)
2+ Identifier les bascules qui ont un effet sur les colonnes analysées en 1+
(**I** (ligne/colonne)=1)
Ces bascules sont considérées comme potentiellement vivantes au cycle C
+ Sauvegarde de l'information dans ***BPVC***.
+ Mise à jour de ***LISTC*** : ***LISTC*** devient l'ensemble des colonnes relatives aux bascules identifiées dans 2+
+ Initialisation de la matrice d'impacts à zéro
Pour i **allant de** C-1 **à** 0 (pas=-1) **faire**

3+ Déterminer, pour chaque colonne, l'ensemble des valeurs de la fonction I
4+ Identifier les bascules qui ont un effet sur les colonnes analysées en 3+
Ces bascules sont considérées comme potentiellement vivantes au cycle i
+ Sauvegarde de l'information dans ***BPVC***.
+ Mise à jour de ***LISTC*** : ***LISTC*** devient l'ensemble des colonnes relatives aux bascules identifiées dans 4+
+ Initialisation de la matrice d'impacts à zéro

Pour j **allant de** 0 **à** nbr_bascules (pas=1) **faire**

Durée_de_vie_bascule_j = 0
Pour k **allant de** 0 **à** nbr_cycles (pas=1) **faire**

Si (la bascule j est parmi celles qui sont potentiellement vivantes au cycle k (***BPVC(k)***) **alors**
+ Incrémentation de *Durée_de_vie_bascule_j*

$$Criticité\_bascule\_j = \frac{Durée\_de\_vie\_bascule\_j}{\text{nbr\_cycles}}$$

Figure IV-15 : Algorithme de calcul de durées de vie.

A chaque cycle d'horloge, l'algorithme n'évalue pas tous les cônes logiques du circuit. En fait, il traite seulement les cônes pouvant causer des erreurs aux instants d'observation. Conséquemment, l'algorithme se contente de déterminer des éléments bien précis dans la matrice d'impacts. Ceci réduit considérablement le temps d'analyse et permet de focaliser l'attention sur les bascules les plus sensibles.

IV.6. Validation expérimentale

Pour mettre en place l'environnement d'analyse de durées de vie présenté dans les sections précédentes, un programme a été développé en C++. Dans l'état actuel, ce programme compte approximativement 5500 lignes de code source. L'objectif principal est de démontrer la faisabilité de l'approche et d'évaluer l'impact des hypothèses faites, notamment l'absence de prise en compte systématique des masquages logiques. Des améliorations en termes de performances restent envisageables parmi les perspectives du travail.

Dans l'intention d'évaluer l'efficacité de l'outil développé, les résultats donnés par ce dernier sont comparés à ceux obtenus par la technique d'injection de fautes, dans le cas d'inversions de bits uniques (SEU). Les analyses comparatives ont été réalisées sur différents exemples de circuits purement matériels, incluant des accélérateurs de chiffrement (appelés aussi crypto-processeurs) et sur un système matériel/logiciel basé sur le processeur LEON3 exécutant des applications variées. Pour les blocs matériels, seuls les plus significatifs sont présentés dans ce document. Les crypto-processeurs utilisés représentent deux implémentations de l'algorithme AES, qui est le standard actuel de chiffrement symétrique sélectionné à la suite du concours lancé par le *National Institute of Standards and Technology (NIST)* en 2001. La première implémentation appelée "AES Parity" [Berto. 03] est équipée d'un système de calcul de parité permettant la détection de fautes. Quant à la deuxième implémentation nommée "AES Morph" [Maist. 13], elle a été renforcée à l'aide d'une contre-mesure de redondance matérielle avec sélection aléatoire face aux attaques par injection de fautes ou par observation (analyse de consommation ou d'émissions électromagnétiques). Le tableau IV-1 résume les caractéristiques globales des trois circuits étudiés. Ces circuits ont été choisis pour avoir une complexité plus significative que dans le cas des circuits benchmark plus traditionnels. Pour les circuits AES, le paramètre nombre de cycles exprime le temps nécessaire pour effectuer un seul chiffrement. Pour le LEON3, ce paramètre dépend évidemment de l'application exécutée. Comme dans le chapitre précédent, nous nous intéressons particulièrement pour le LEON3 aux éléments de mémorisation dans la logique aléatoire au niveau du pipeline entier.

Tableau IV-1 : Caractéristiques des circuits analysés.

Circuit	Nombre d'entrées primaires	Nombre de sorties primaires	Fréquence de fonctionnement	Nombre de cycles	Nombre de bascules	Nombre d'instances combinatoires
AES Parity	165	38	50 Mhz	35	945	34499
AES Morph	268	131	100 Mhz	74	2328	36718
Pipeline LEON3	988	1286	50 Mhz	-	988	111019

Des campagnes exhaustives d'injection de fautes ont été réalisées sur les deux crypto-processeurs pendant l'exécution d'une série de chiffrements. Pour le LEON3, vu la complexité du circuit et le grand nombre d'injections possibles, nous avons utilisé les résultats de campagnes statistiques. Le nombre de fautes injectées est défini selon les études faites dans [Leve. 09]. Dans tous les cas, les fautes injectées sont de multiplicité spatiale de 1 bit (SEU) et la

technique d'injection utilisée est celle basée sur l'émulation matérielle [WW. 4]. Une étude comparative est faite entre les résultats de ces campagnes et ceux donnés par l'outil d'analyse de durées de vie. Cette comparaison tient principalement compte des critères suivants : (1) cohérence des résultats et (2) durée des expérimentations.

IV.6.1. Cohérence des résultats

Les figures IV-16 à IV-20 illustrent, pour chaque cas d'étude, les pourcentages de criticité obtenus par les deux approches d'évaluation. Pour le cas du processeur LEON3, les résultats sont présentés pour les registres ayant un impact fort sur l'exécution des instructions (193 bascules, soit environ 20% des bascules de l'unité entière).

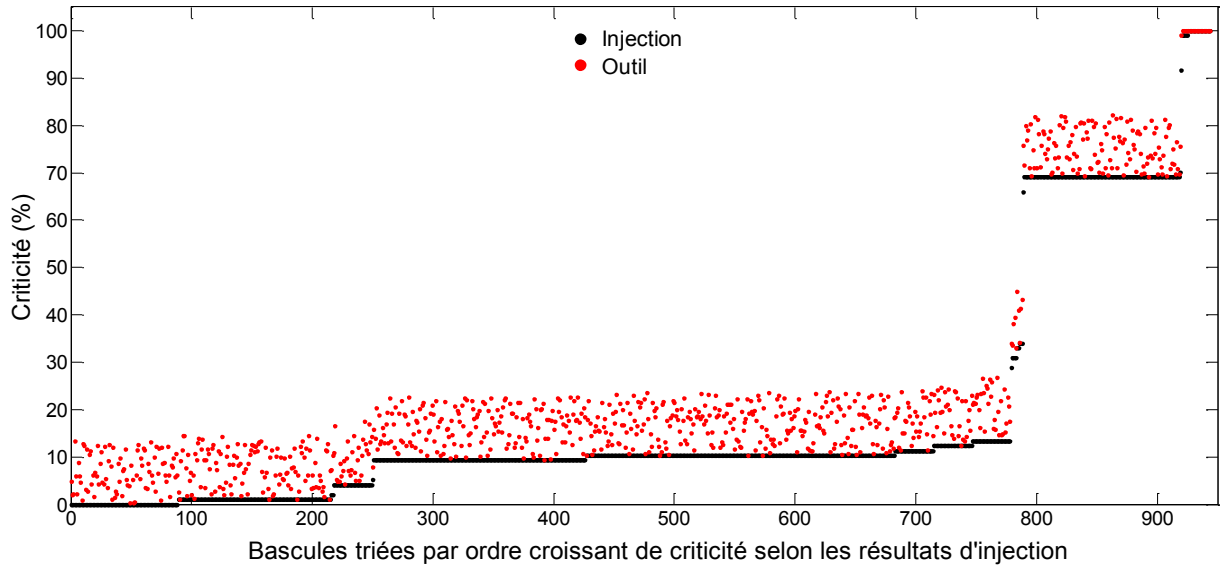


Figure IV-16 : Comparaison des résultats de criticité par bascule pour le circuit AES Parity.

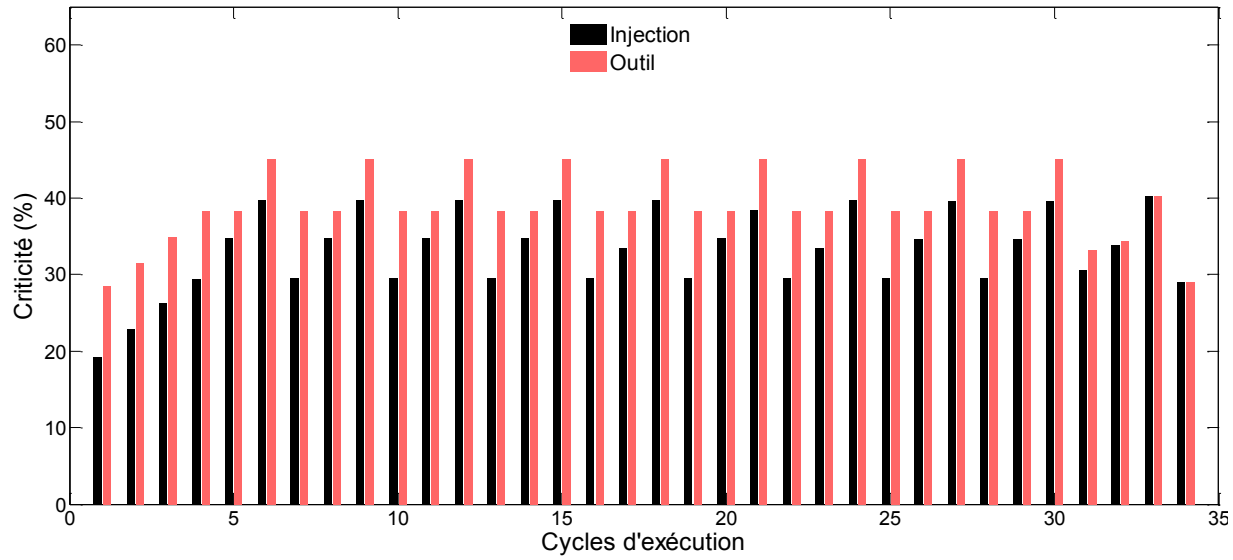


Figure IV-17 : Comparaison des résultats de criticité par cycle pour le circuit AES Parity.

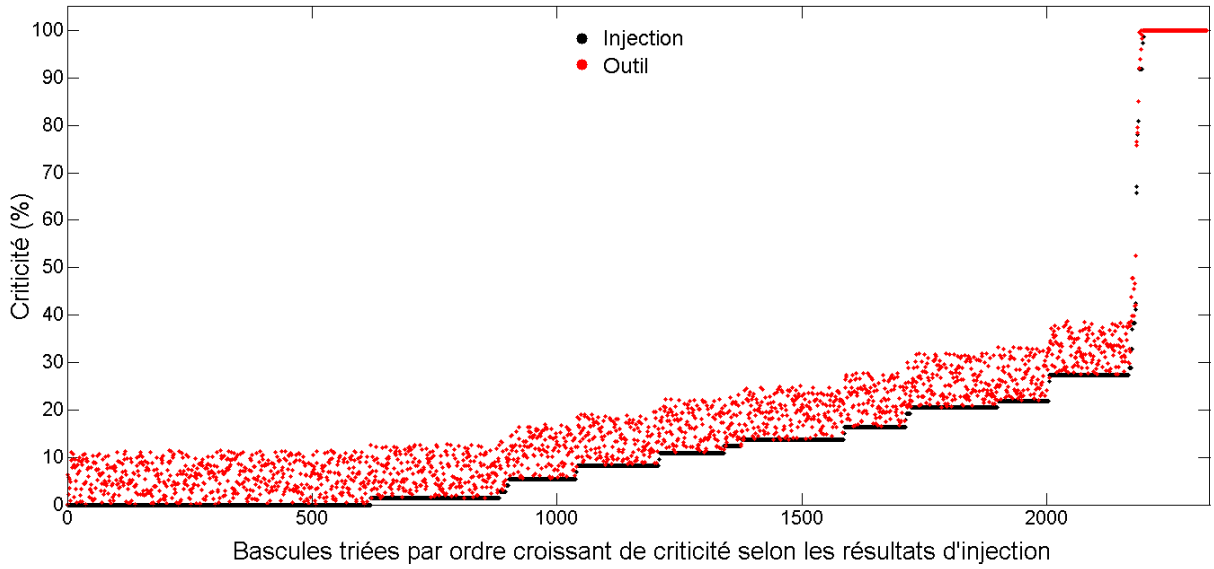


Figure IV-18 : Comparaison des résultats de criticité par bascule pour le circuit AES Morph.

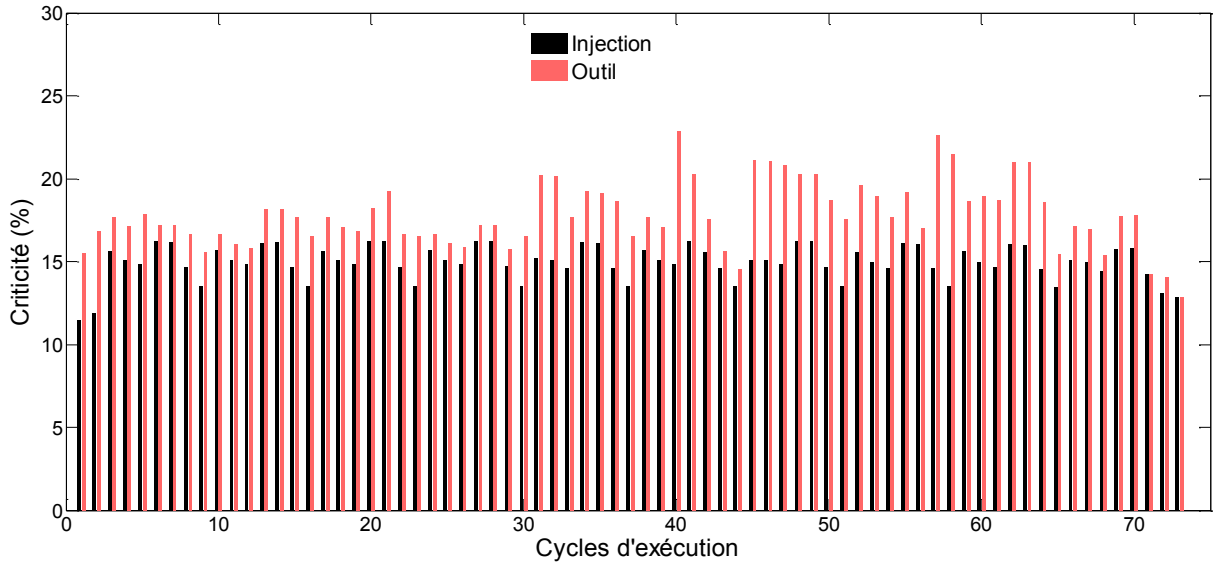


Figure IV-19 : Comparaison des résultats de criticité par cycle pour le circuit AES Morph.

Les expériences exhaustives d'injection de fautes effectuées sur les deux exemples de crypto-processeurs ont permis d'obtenir des valeurs exactes de criticité. La comparaison de ces résultats de référence avec les résultats donnés par l'outil d'évaluation montre une très bonne corrélation entre les deux approches, que ce soit en termes de criticité par bascule ou par cycle (figures IV-16 à IV-19). Pareillement, pour les différentes applications exécutées sur le LEON3, les résultats obtenus par l'outil sont très cohérents avec ceux issus des injections statistiques (figure IV-20). D'ailleurs, comme le confirme le tableau IV-2, la marge d'erreur entre les valeurs de criticité globale ne dépasse pas les 8% dans tous les cas ce qui est acceptable, surtout que le nombre d'injections choisi ne garantit qu'une précision avec 5% de marge d'erreur pour un taux de confiance de 95%. Il faut noter que les campagnes de qualification sous faisceau de particules ont souvent des marges d'erreur nettement plus grandes.

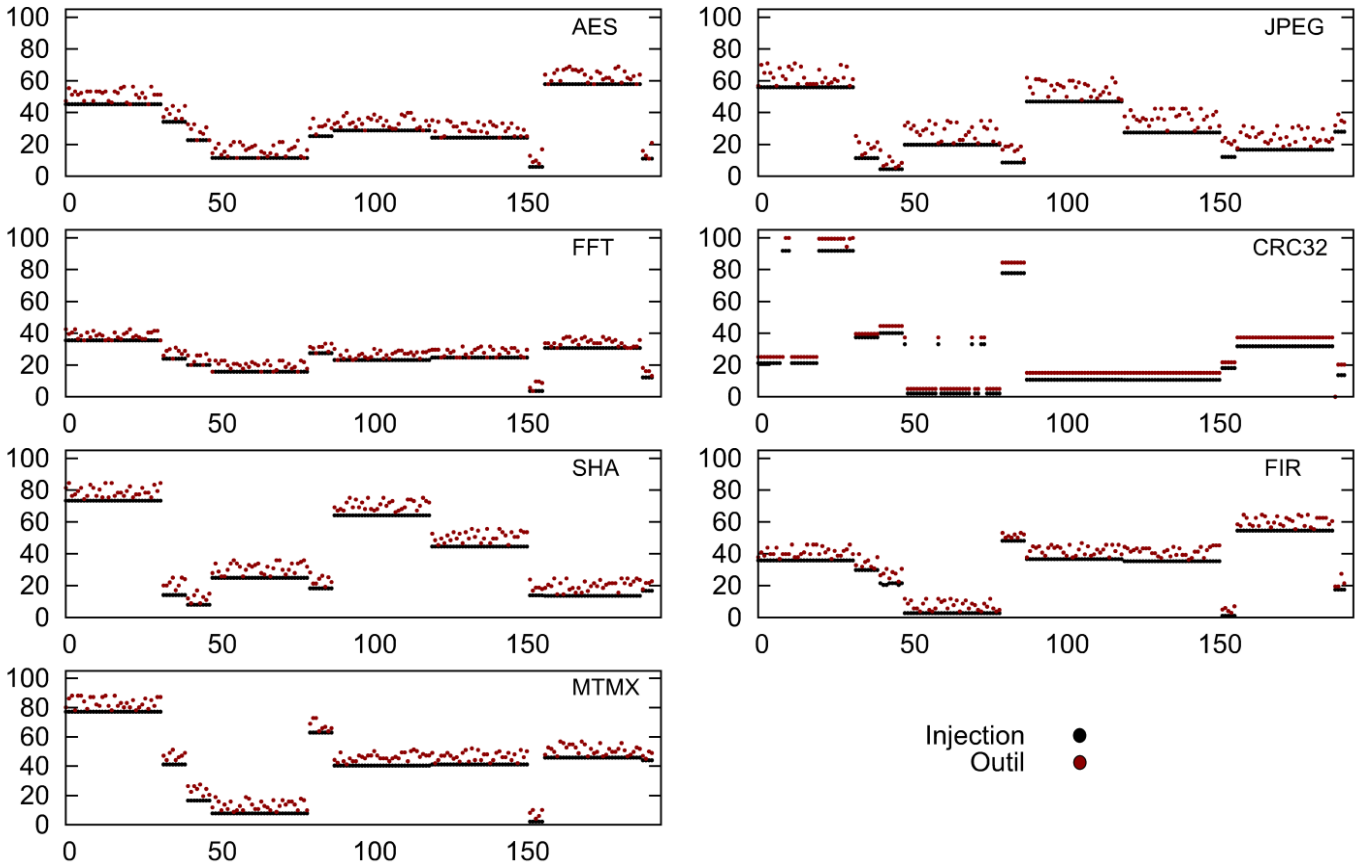


Figure IV-20 : Comparaison des résultats de criticité des bascules internes pour les applications CRC32, AES, FFT, JPEG, SHA, FIR et MTMX.

Tableau IV-2 : Comparaison des résultats de criticité globale.

Système	Injections (%)	Outil (%)	Marge d'erreur (%)
AES Parity	18,6	25,1	6,5
AES Morph	14,5	20,3	5,8
LEON3 – AES	26,49	29,82	3,3
LEON3 – FFT	21,75	24,67	2,9
LEON3 – JPEG	28,35	35,64	7,29
LEON3 – CRC32	36,57	41,10	4,5
LEON3 – MTMX	38,55	44,42	5,9
LEON3 – SHA	31,69	37,59	5,9
LEON3 – FIR	28,56	33,64	5,1

Il faut aussi remarquer que les valeurs de criticité fournies par l'outil d'analyse de durées de vie sont toujours supérieures à celles données par les campagnes d'injection de fautes. Cette surestimation correspond au fait que l'outil fait une analyse "pire cas" en ignorant certaines situations de masquage logique dans les réseaux combinatoires séparant les éléments de mémorisation. Toutefois, la précision obtenue est suffisante pour faire des choix cohérents lors de la sélection d'un sous-ensemble d'éléments critiques pour une redondance sélective. Il est en effet remarquable, sur les figures précédentes, que les évaluations de l'outil suivent précisément les variations fortes de criticité obtenues avec les injections de fautes.

IV.6.2. Durée des expérimentations

Il est important, au-delà de la cohérence des résultats, de voir ce que le nouvel outil apporte au niveau expérimental par rapport à la technique d'injection de fautes. Le tableau IV-4 résume, pour les deux approches, le temps moyen nécessaire pour calculer la criticité des différentes bascules. La machine sur laquelle l'outil a été exécuté est dotée d'un processeur Intel Core 2 Duo cadencé à 3,16 GHz avec 4 Go de mémoire vive. Les campagnes d'injection ont été effectuées, pour chaque cas de circuit, sur un prototype implanté sur une carte d'évaluation embarquant un FPGA Virtex V [WW. 2]. Les temps indiqués correspondent uniquement au déroulement de l'analyse ; ils ne prennent pas en compte le temps requis pour mettre en place les expériences d'injection sur la carte FPGA (synthèse et placement du circuit, adaptation des logiciels, etc.).

Vu les résultats du tableau IV-3 et bien que l'outil soit un premier prototype, il s'avère que notre flot d'analyse offre un gain sensible en temps de calcul par rapport à l'injection par émulation, permettant de réaliser davantage d'itérations pendant le développement. Il faut noter que ce flot permet de réutiliser directement l'environnement de validation fonctionnelle mis en place pour le circuit, et qu'il est beaucoup plus accessible aux développeurs puisqu'il ne nécessite pas d'outils ou de plateforme matérielle spécifique.

Tableau IV-3: Temps moyen pris pour l'évaluation de criticité.

Circuit	Nombre d'injections	Emulation (min)	Simulation/Outil (min)	Facteur d'accélération
AES Parity	33075	108	16 total	6,75
AES MORPH	172272	169	33 total	5,12
LEON3 – FIR	98783	287	5/80	3,38
LEON3 – MTMX	124355	389	2/13	25,93
LEON3 – CRC32	304623	502	7/50	8,81
LEON3 – AES	520589	670	27/202	2,93
LEON3 – SHA	575773	710	19/224	2,92
LEON3 – FFT	917042	820	137/350	1,68
LEON3-JPEG	1019539	990	248/488	1,35

En ce qui concerne les signaux suivis en simulation fonctionnelle, le tableau IV-4 indique que leur nombre est très négligeable par rapport au nombre total de signaux, de sorte que l'impact sur le temps de simulation est aussi négligeable. A titre d'exemple, dans le cas du circuit AES Parity, il a fallu obtenir la trace de simulation de seulement 32 signaux pour parvenir à obtenir des résultats de criticité similaires à ceux issus des injections.

Tableau IV-4 : Evaluation du nombre de signaux suivis en simulation.

Circuit	Nombre total de signaux	Nombre de signaux suivis en simulation	Pourcentage de signaux suivis
AES Parity	8556	32	0,37 %
AES Morph	25054	47	0,19 %
Pipeline LEON3	228820	550	0,24%

IV.7. Comparaison avec l'algorithme de prédiction de criticité des registres

Pour le cas du processeur LEON3, nous faisons une comparaison entre l'outil et l'approche analytique présentée dans le chapitre précédent. Le but ici n'est pas d'identifier les bascules les plus critiques, mais de faire une synthèse des méthodes développées au cours de cette thèse. Les applications concernées sont : AES, FFT, JPEG, CRC32 et SHA qui ont également été utilisées dans le chapitre précédent. La comparaison pointe sur deux aspects : la précision et la rapidité de l'analyse de criticité. La comparaison avec les injections de fautes est également montrée.

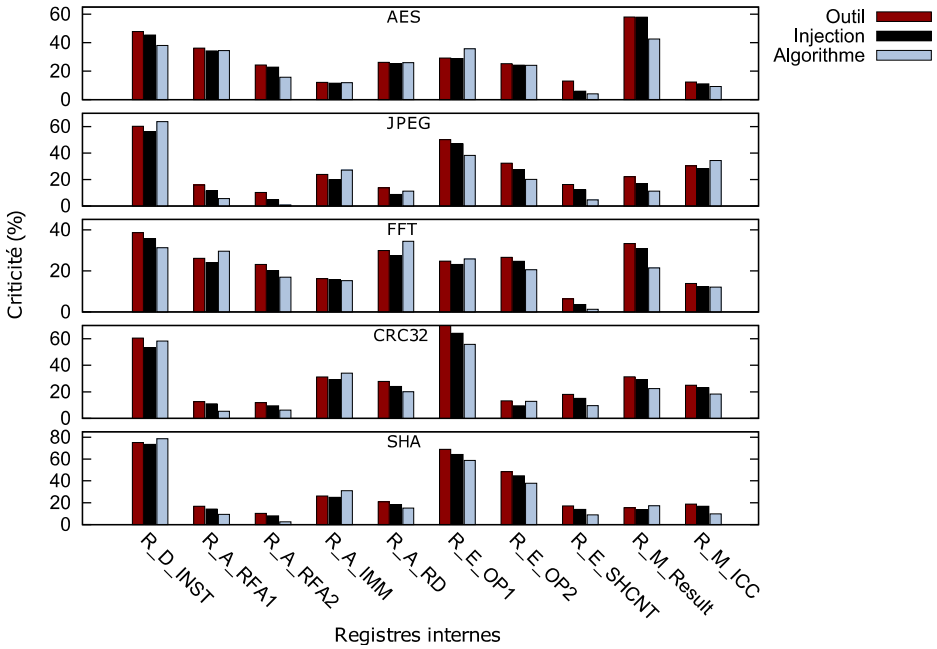


Figure IV-21 : Comparaison des résultats de criticité des registres internes donnés par les trois approches.

La figure IV-21 montre les résultats d'évaluation de criticité des registres internes du LEON3 tels que déterminés par les trois méthodes d'évaluation de robustesse. Dans tous les cas de figure, nous remarquons que les valeurs de criticité fournies par l'outil sont toujours supérieures à celles données par les campagnes d'injection. Celles calculées par l'algorithme de prédiction sont tantôt supérieures, tantôt inférieures aux résultats d'injection donc ne peuvent pas être considérées comme un "pire cas" mais plutôt comme un ordre de grandeur.

Une des différences principales entre les deux méthodes proposées réside dans le fait que l'outil effectue une analyse de durées de vie par bascule alors que l'algorithme de prédiction a été conçu pour évaluer la sensibilité de groupements de bascules (registres). En effet, les bits constituant un même registre peuvent avoir des valeurs de criticité différentes. Cela dépend de leur utilisation par la logique combinatoire. Par exemple, pour les 32 bascules formant le registre d'instruction de l'étage *Decode*, l'outil qualifie certains bits comme plus critiques que d'autres. La figure IV-22 illustre la criticité des bits de ce registre pour les applications AES et JPEG. Nous constatons que les bascules les moins critiques correspondent aux bits inutilisés dans le cas de l'instruction NOP (bits 29-24 et 21-0). Ceci affermit en quelque sorte l'hypothèse retenue dans l'algorithme de prédiction pour exclure l'instruction NOP du calcul de criticité du registre D_INST (dans un contexte de fautes de type SEU). Mais cela explique aussi une partie de la sous-estimation de criticité obtenue avec l'algorithme.

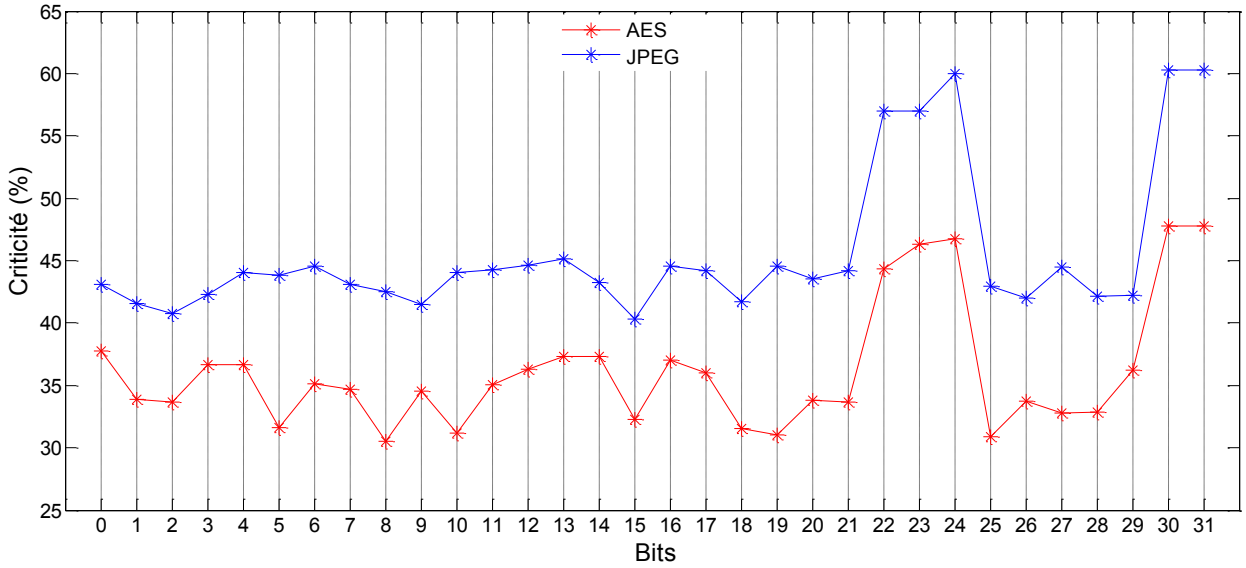


Figure IV-22 : Résultats de criticité des bits du registre d'instruction D_INST, obtenus avec l'analyse de durées de vie.

Il a été démontré précédemment que les deux méthodes développées offrent un gain en matière d'évaluation rapide de robustesse par rapport à l'émulation qui est déjà une approche d'injection de fautes très optimisée. Le tableau IV-5 indique que l'algorithme de prédiction se démarque par un facteur d'accélération nettement supérieur à celui de l'outil d'analyse de durées de vie. Le manque de rapidité de la part de l'outil est cependant compensé par une analyse plus fine de la sensibilité des registres du LEON3. En outre, l'outil permet de faire une évaluation plus complète en mesurant la criticité de tous les points mémoires répartis sur les 7

étages du pipeline, tandis que les modèles d'instructions élaborés pour l'algorithme de prédiction n'incluent qu'un ensemble précis de registres internes.

Tableau IV-5 : Gains des deux approches proposées par rapport à la technique d'injection de fautes par émulation.

Application	Gain (Algorithme de prédiction/Injection)	Gain (Outil/Injection)
AES	45	2,93
FFT	48	1,68
JPEG	55	1,35
CRC32	38	8,81
SHA	49	2,92

En fonction de ces résultats, il est possible de compléter le tableau de synthèse présenté au chapitre III. Le tableau IV-6 résume la comparaison entre l'ensemble des approches utilisées dans notre travail, dans le cas d'un système à base de microprocesseur. Pour les blocs purement matériels, seules les deux premières approches sont utilisables et l'évaluation de durées de vie proposée dans ce chapitre apporte une alternative réelle aux injections de fautes.

Tableau IV-6 : Comparaison des quatre approches d'évaluation de robustesse.

Approches	Précision des évaluations	Rapidité d'analyse et contraintes
Injection de fautes (émulation)	✔ ✔ ✔	✘
Evaluation de durées de vie	✔ ✔	✔
Approche analytique (algorithme de prédiction)	✔	✔ ✔
Evaluations lors de la compilation	✘	✔ ✔ ✔

IV.8. Conclusion

Dans ce chapitre, nous avons décrit une méthode générique d'évaluation de robustesse basée sur l'analyse de durées de vie. Cette analyse repose sur l'utilisation des informations fournies par l'étape d'élaboration et des traces de simulation de certains signaux clés.

Trois modules résument le flot d'analyse proposé. Le premier consiste à extraire les différents cônes logiques du circuit. Le deuxième module a pour mission d'identifier les signaux clés qui contrôlent la propagation des données. Ces signaux sont par la suite suivis pendant la simulation fonctionnelle du circuit et leurs traces d'exécution sont enregistrées. Le dernier module exploite les résultats des deux modules précédents pour calculer les durées de vie et déduire le niveau de criticité de chaque bascule. Nous soulignons le fait que pour un circuit donné, l'étape d'extraction des cônes et celle d'identification des signaux clés sont appelés une seule fois. L'analyse de criticité dépend ensuite des traces obtenues pour une exécution donnée. Ceci rend possible des évaluations itératives pour différents scénarios d'application.

La méthode a été entièrement automatisée et des expériences d'injection de fautes par émulation sur des circuits différents ont été réalisées afin d'évaluer son efficacité. La comparaison des résultats a bien montré une cohérence avec une marge d'erreur satisfaisante. L'approche proposée est plus avantageuse que l'injection de fautes par sa rapidité de calcul. De plus, contrairement à l'injection par émulation qui nécessite la synthèse logique et l'implémentation sur une carte de développement, notre méthode n'a besoin que de quelques traces de simulation pour évaluer la criticité, ce qui réduit notablement les coûts d'expérimentation et les compétences requises pour réaliser l'évaluation. Enfin, la comparaison avec l'approche présentée dans le chapitre 3 nous a permis de faire le point sur les avantages et les inconvénients des approches proposées dans le cadre de cette thèse.

Il faut cependant rappeler que notre approche ne permet aujourd'hui d'évaluer que la sensibilité intrinsèque d'un circuit ; nous reviendrons sur ce point dans les perspectives de nos travaux.

Conclusion générale et perspectives

L'évolution des technologies de semi-conducteurs, notamment la réduction des dimensions, favorise les perturbations externes des circuits. Ces perturbations d'origines multiples, naturelles ou intentionnelles, peuvent conduire à des comportements erronés qui sont intolérables, surtout pour des circuits utilisés dans des systèmes critiques. En conséquence, la conception des circuits intégrés numériques exige de plus en plus de techniques et d'outils pour l'évaluation de leur niveau de robustesse. Ceci permet de quantifier le risque de défaillance sous certaines conditions et de réduire les coûts en focalisant les protections sur les blocs les plus critiques vis-à-vis des perturbations.

Au cours de cette thèse, nous avons opté pour des analyses de robustesse tôt dans le flot de conception afin de quantifier la sensibilité globale lors de l'exécution d'une application donnée et d'identifier les éléments les plus critiques. Dans ce contexte, deux nouvelles approches ont été proposées.

La première approche est destinée aux systèmes à base de microprocesseur et permet d'évaluer la criticité des registres architecturaux situés dans le banc de registres et des registres internes contenus dans les étages de pipeline de la microarchitecture. L'approche a été démontrée sur l'exemple d'un processeur SPARC v8 (LEON3), en développant un algorithme de prédiction de criticité spécifique pour son pipeline de calcul sur entiers. Cet algorithme (ou modèle micro-architectural) a permis de faire des analyses de criticité en fonction de plusieurs paramètres tels que les options de compilation du logiciel et les paramètres architecturaux du processeur LEON3.

La deuxième approche est plus générique et permet d'analyser la robustesse intrinsèque de tout circuit numérique synchrone sans hypothèse spécifique d'architecture. Elle s'applique aussi bien à un microprocesseur exécutant un programme qu'à un bloc purement matériel. L'outil mettant en œuvre cette approche permet de faire l'analyse des durées de vie de tous les registres internes, puis de calculer la criticité des différents points mémoire, des différents cycles d'exécution ou la sensibilité globale d'une application donnée à partir d'une description synthétisable et de la définition de l'application (définition des valeurs des entrées/sorties et éventuellement logiciel embarqué). Il a été éprouvé sur plusieurs circuits de complexité significative comme des crypto-processeurs et un système utilisant le processeur LEON3.

Afin de valider ces approches, des campagnes d'injection de fautes par émulation ont été effectuées sur les circuits étudiés. La comparaison des résultats entre nos approches et les injections de fautes a montré que nos approches ont une précision comparable aux injections de fautes statistiques usuelles et permettent de bien identifier les registres internes les plus critiques pour une application donnée. Elles se distinguent par leur rapidité de calcul et la réutilisation de l'environnement mis en place pour les validations fonctionnelles, sans nécessiter un matériel ou des compétences spécifiques. Les itérations d'optimisation de la robustesse intrinsèque, par exemple par modification du logiciel embarqué, sont ainsi grandement

facilitées. Il a été montré que ces itérations peuvent être contrôlées beaucoup plus précisément avec nos approches qu'avec des approches basées sur des métriques obtenues lors de la compilation.

L'accès à des signaux internes pendant la simulation peut ne pas être possible pour des circuits dont la description matérielle est obfusquée. Nos approches ne sont alors pas applicables. Mais cet inconvénient est partagé avec les approches d'injection qui ont besoin de modifier ou tout du moins de connaitre l'emplacement des cellules logiques ciblées. L'autre limitation majeure de l'approche présentée pour les microprocesseurs est le temps requis pour développer le modèle d'analyse, représentatif de la microarchitecture du circuit. Ceci n'est rentable que pour des circuits utilisés dans de nombreux systèmes (comme le LEON3). Il est difficile d'envisager de construire un tel modèle pour un circuit qui ne sera utilisé que dans un système donné et dans des conditions bien connues. La seconde approche proposée est alors une réponse, car elle ne nécessite que la disponibilité de la description synthétisable, le flot d'analyse étant complètement automatisé sous réserve d'accepter les quelques limitations indiquées dans le document.

Cette seconde approche a toutefois encore, à ce stade, des limitations. Outre un temps d'analyse plus élevé, elle est incapable, dans l'état actuel, d'identifier des structures particulières qui sont implantées pour éviter la propagation des erreurs. Par exemple, une architecture tripliquée (TMR) ne peut pas être analysée comme étant plus robuste qu'une architecture non protégée, puisque les voteurs ne seraient vus que comme des éléments combinatoires indifférenciés, correspondant à des chemins potentiels de propagation d'erreur. Comme indiqué à plusieurs reprises, l'approche ne peut actuellement évaluer avec de bons résultats que la robustesse intrinsèque d'un circuit sans protection spécifique contre les erreurs.

Ce travail de thèse offre des perspectives variées :

L'algorithme de prédiction peut être étendu ou être rendu configurable pour inclure les cas de fautes multiples telles que les MBUs et les MCUs pouvant affecter plusieurs points mémoire à la fois. Dans un premier temps, une comparaison des analyses obtenues avec des injections de fautes multiples devra permettre d'évaluer la précision actuelle dans ce type de situation. Bien sûr, l'approche peut aussi être appliquée à d'autres microprocesseurs, notamment ceux destinés plus particulièrement aux systèmes embarqués devant fonctionner dans des environnements agressifs.

En ce qui concerne l'outil d'analyse de durées de vie, les résultats pourront aussi être comparés avec des résultats provenant de campagnes d'injection de fautes multiples. L'outil dans son état actuel n'est qu'un premier prototype, qui peut être davantage optimisé pour améliorer le facteur d'accélération par rapport à l'injection par émulation. Il peut évoluer aussi pour prendre en compte les blocs mémoire ou les circuits multi-horloges et certaines caractéristiques comme la validation d'horloge. Il peut aussi évoluer pour être capable, dans une structure après élaboration, de reconnaître certaines fonctions logiques particulières. Sans prendre en compte tous les cas de masquage logique potentiels, il serait alors possible de

prendre en compte par exemple une structure typique comme une triplication avec vote majoritaire.

Au-delà des améliorations ou extensions directes des approches proposées, une autre perspective importante concerne la prise en compte des contraintes système. Pour nos approches, comme dans la plupart des cas d'injections de fautes, les sorties erronées d'un circuit sont considérées comme conduisant à une défaillance. Or des études antérieures ont montré que des sorties erronées, même avec une grande variation sur la valeur ou une durée correspondant à nombreux cycles, ne conduisent pas forcément à un évènement critique pour le système global. Dans certains cas, le déterminisme des opérations n'est même pas fondamental. Par exemple, un routeur internet qui inverse l'ordre d'émission de deux trames correspondant à deux messages différents, mais qui les transmet aux bonnes destinations, aura des sorties erronées par rapport à la spécification mais n'induira pour les utilisateurs qu'un très petit décalage temporel, peu perceptible. Aujourd'hui, des environnements sophistiqués de vérification fonctionnelle permettent de modéliser (jusqu'à un certain point) ces cas particuliers qui restent acceptables du point de vue du système, même s'ils ne correspondent pas exactement à la spécification. Nos approches exploitent aujourd'hui seulement une trace de simulation fonctionnelle pour évaluer la criticité des erreurs. Le couplage de nos approches avec ces environnements sophistiqués de vérification fonctionnelle pourrait permettre d'aller plus loin dans l'analyse et d'affiner l'identification des cas où une sortie erronée du circuit va effectivement devenir critique pour le système. Un tel couplage n'est cependant pas trivial et nécessitera un travail conséquent.

Bibliographie

[Agoy. 10] M. Agoyan, J. Dutertre, D. Naccache, B. Robisson, "When Clocks Fail : On Critical Paths and Clock Faults", Smart Card Research and Advanced Application Conference (CARDIS), Springer Berlin Heidelberg , T. 6035, Lecture Notes in Computer Science, pp. 182-193, Passau, Germany, April 2010.

[Agui. 05] M.A. Aguirre, J.N. Tombs, F. Muñoz, V. Baena, A. Torralba, A. Fernández-León, F. Tortosa-López, "FT-UNSHADES: A new system for SEU injection, analysis and diagnostics over post synthesis netlist", in proceedings NASA Military and Aerospace Programmable Logic Devices, MAPLD, Washington, D.C., September 2005.

[Alex. 02] D. Alexandrescu, L. Anghel and M. Nicolaidis, "New methods for evaluating the impact of single event transients in VDSM ICs", Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on, 2002, pp. 99-107.

[Alex. 04] Alexandrescu, L. Anghel, M. Nicolaidis, "Simulating Single Event Transients in VDSM ICs for Ground Level Radiation", Journal of Electronic Testing: Theory and Applications (JETTA), pp. 413-421, Août 2004.

[And. 97] R. Anderson, M. Kuhn, "Low cost attacks on tamper resistant devices", Proceedings of 5th International Workshop on Security Protocols, pp. 125-136, 1997.

[Ane. 00] G.M. Anelli, "Conception et caracterisation de circuits integres resistants aux radiations pour les detecteurs de particules du LHC en technologies CMOS sub-microniques profondes", Thèse de doctorat, Institut National Polytechnique de Grenoble, Novembre 2000.

[Angh. 00] Anghel L., "Les Limites Technologiques du Silicium et Tolérance aux Fautes", Thèse de Doctorat, Laboratoire TIMA, INPG Grenoble, 15 Décembre 2000.

[Anton. 00] L. Antoni, R. Leveugle, B. Fehér, "Using run-time reconfiguration for fault injection in hardware prototypes", IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 405-413.

[Anton. 03] L. Antoni, R. Leveugle, B. Fehér, "Using run-time reconfiguration for fault injection applications", IEEE Transactions on Instrumentation and Measurement, vol. 52, no. 5, October 2003, pp. 1468-1473.

[Arlat. 94] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, D. Powell, " Fault Injection For Dependability Validation - A Methodology and Some Applications', IEEE Trans. on Software Engineering, vol. 16, pp. 166–182, Février 1990.

[Baraz. 00] J. C. Baraza, J. Gracia, D. Gil and P. J. Gil, "A prototype of a VHDL-based fault injection tool," *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*, Yamanashi, 2000, pp. 396-404.

[Bay. 12] P. Bayon, L. Bossuet, A. Aubert, V. Fischer, F. Poucheret, B. Robisson, P. Maurine, "Contactless Electromagnetic Active Attack on Ring Oscillator Based True Random Number Generator", Constructive Side-Channel Analysis and Secure Design (COSADE), 2012.

[Benj. 13] Ben-Jrad M., "Robustesse par conception de circuits implantés sur FPGA SRAM et validation par injection de fautes", Thèse de Doctorat, Laboratoire TIMA, INPG Grenoble, Juillet 2013.

[Bens. 00] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ Source to Source Compiler for Dependable Applications", IEEE Asian Test Symposium (ATS 2001), Kyoto (J), November 2001, pp.209-303

[Berg. 10] S. Bergaoui, P. Vanhauwaert, and R. Leveugle, "A New Critical Variable Analysis in Processor-Based Systems," IEEE Trans. Nucl. Sci., vol. 57, no. 4, pp. 1992-1999, Aug. 2010.

[Berro. 02] L. Berrojo, I. Gonzales, F. Corno, M. Sonza-Reorda, G. Squillero, L. Entrena, C. Lopez, "New Techniques for Speeding Up Fault Injection Campaigns", Design, Automation and Test in Europe Conference (DATE "02), Paris, France, pp. 847-852, Mars 2002.

[Berto. 03] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard", IEEE Transactions on Computers, vol. 52, no. 4, pp. 492–505, 2003.

[Bisw. 05] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures", In International Symposium on Computer Architecture (ISCA) 2005.

[Blome. 06] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in Proc. CASES, 2006, pp. 421–431.

[Bolch. 10] C. Bolchini, C. Sandionigi. "Fault Classification for SRAM-Based FPGAs in the Space Environment for Fault Mitigation", IEEE Embedded Systems Letters, vol. 2, no 4, pp. 107-110, 2010.

[Boneh. 97] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", Advances in Cryptology, Proceedings of EUROCRYPT "97, Springer-Verlag LNCS 1233, pp. 37-51, 1997.

[Bosio. 08] Bosio, A., et G. Di Natale. 2008. " LIFTING: A Flexible Open-Source Fault Simulator ", In ATS "08. 17th. (24-27 Nov. 2008), p. 35-40.

[Boud. 95] J. Boudenot, "L'Environnement Spatial", Collection "Que sais-je ?", Ed. Presses Universitaires de France, 1995.

[Boud. 99] J. Boudenot, "Tenue des circuits aux radiations ionisantes", volume E 3 950. Techniques de l'Ingénieur.

[Boue. 98] J. Boue, P. Petillon, Y. Crouzet, "MEFISTO-L: a VHDL-based Fault Injection Tool for the Experimental Assessment of Fault Tolerance",28th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-28), Munich, Allemagne, pp. 168-173, Juin 1998.

[Buch. 97] S. Buchner, M. Olmos, P. Cheynet, R.Velazco, D. McMorrow, J. Melinger, R. Ecoffet, J. D. Muller, "Pulsed laser validation of recovery mechanism of critical SEEs in artificial network system", Proceedings of 4th European Conference on Radiation and its Effects on Component and Systems (RADECS'97), pp.110-111, Cannes, France, 1997.

[Caniv. 10] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, M. Renaudin, "Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA", Springer-Verlag, Journal of Cryptology, Vol. 24, No. 2, pp. 247-268, 2010.

[Cha. 96] H. Cha, E.M. Rudnick, J.H. Patel, R.K. Iyer, G.S. Choi, "A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults", IEEE Trans. on Computers, vol. 45, issue 11, pp. 1248-1256, Novembre 1996.

[Cive. 01a] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Exploiting FPGA for accelerating fault injection experiments", Proc. 7th Int. On Line Testing Workshop (IOLTW "01), Taormina, Italie, pp. 9-13, Juillet 2001.

[Cive. 01b] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "FPGAbased Fault Injection for Microprocessor Systems", 10th Asian Test Symp. (ATS'01), Kyoto, Japon, p. 3004, Novembre 2001.

[Cive. 03] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante,"New Techniques for Efficiently Assessing Reliability of SoCs", Microelectronics Journal, vol 34, issue 1, pp. 53-61, Janvier 2003.

[Corn. 00] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "RT-Level Fault Simulation Techniques based on Simulation Command Scripts", Proc. XV Conf. on Design of Circuits and Integrated Systems (DCIS'00), Montpellier, France, Novembre 2000.

[Dave. 09] JM. Daveau, G. Gasiot, P. Roche, A. Blampey, J. Bulone, "An Industrial Fault Injection Platform for Soft-Error Dependability Analysis and Hardening of Complex System-on-a-Chip", Proceeding of the 47th IEEE International Reliability Physics Symposium, 2009.

[Dehb. 12] A. Dehbaoui, J. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, A. Tria, "Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system", Rapport technique, IACR Cryptology ePrint Archive, 2012.

[Dehb. 13] A. Dehbaoui, A. Mirbaha, N. Moro, J. Dutertre, A. Tria, "Electromagnetic Glitch on the AES Round Counter", Constructive Side-Channel Analysis and Secure Design (COSADE), 2013.

[Duter. 11] J. Dutertre, Jacques J. A. Fournier, A. Mirbaha, D. Naccache, J. Rigaud, B. Robisson, A. Tria, "Review of fault injection mechanisms and consequences on countermeasures design", 6th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2011.

[Fau. 05] F. Faure, "Injection de fautes simulant les effets de basculement de bits induits par radiation", Thèse de doctorat, Institut National Polytechnique de Grenoble, Novembre 2005.

[Franc. 06] Julien Francq, Pascal Manet, and Jean-Baptiste Rigaud. Material emulation of faults on cryptoprocessors. In Proceedings of Sophia Antipolis forum of MicroElectronics (SAME), 2006.

[Gais. 13] Cobham GAISLER, http://www.gaisler.com/index.php/products/processors/leon3..

[Gasio. 01] G. Gasiot, "Etude de la Sensibilité des Technologies CMOS/SoI et CMOS bulk aux rayonnements radiatifs terrestres", Thèse de doctorat, Université de Bordeaux, Juin 2001.

[Geor. 10] N.J. George, C.R. Elks, B.W. Johnson and J. Lach. Transient fault models and AVF estimation revisited. In Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on, pages 477–486, June 2010.

[Gunn. 89] U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of error Detection Schemes Using Fault Injection by Heavy-Ion Radiation", International Symposium on Fault-Tolerant Computing (FTCS), pp. 340-347, June 1989.

[Guti. 04] D. Gonzales Gutierrez, "Single Event Upsets Simulation Tool Functional Description, ESA document, Juillet 2004.

[Hadj. 05] K. Hadjiat, "Evaluation Predictive de la Surete de Fonctionnement d'un Circuit Integre Numerique", These de Doctorat, Laboratoire TIMA, INPG Grenoble, Juin 2005.

[Hazu. 00] P. Hazucha, C. Svensson, and S. Wender. "Cosmic-Ray Soft Error Rate Characterization of a Standard 0. 6-f.lM CMOS Process", IEEE Journal of Solid-State Circuits, 3 5 : 1 422-1 429, October 2000.

[Henn. 11] John L. Hennessy and David A. Patterson. 2011. Computer Architecture, Fifth Edition: A Quantitative Approach", 5th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[Jedec. 01] JEDEC Standard no. JESD89, "Measurements and reporting of alpha particles and terrestrial cosmic ray-induced soft errors in semiconductor devices", August 2001.

[Jedec. 07] JEDEC Standard, "Test Method for Alpha Source Accelerated Soft Error Rate", JESD89 2A, October 2007.

[Jenn. 94] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models : The MEFISTO tool, In Proceedings of the 24th International Symposium on Fault Tolerant Computing, (FTCS-24), IEEE, Austin, Texas, USA, pages 66–75, 1994.

[Jes. 96] JEDEC Standard, "Test procedure for the measurement of single-event effect in semiconductor devices from heavy ions irradiation", EIAD/JEDEC, 1996.

[Kafk. 06] Leoš Kafka, Ondřej Novak, "FPGA-based Fault Simulator", 9th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), Prague, Republique Tcheque, pp. 218-219, Avril 2006.

[Karls. 91] J. Karlsson, U. Gunneflo, P. Lidén, J. Torin, " Two fault injection techniques to test of fault handling mechanisms', Proc. Test Conference (ITC'91), pp. 140–149, 1991.

[Kay] S. Kayali, "Space Radiation effects on Microelectronics", NASA Jet Propulsion Laboratory, http://parts.jpl.nasa.gov./docs/Radcrs_Final.pdf.

[Kog. 98] R. Koga, "Single event functional interrupt (SEFI) sensitivity in EEPROMs", Proceedings of MAPLD, pp. 2-12, Greenbelt, Maryland, USA, 1998.

[Lapr. 04] J. Laprie, "Sûreté de fonctionnement informatique: concepts, défis, directions", ACI Sécurité et Informatique, Toulouse, Novembre 2004.

[Lee. 09] J. Lee, A. Shrivastava, "Static analysis to mitigate soft errors in register files", Design, Automation and Test in Europe Conference (DATE), pp. 1367-1372, 2009.

[Leve. 00] R. Leveugle and K. Hadjiat. Optimized generation of vhdl mutants for injection of transition errors. In SBCCI "00 : Proceedings of the 13th symposium on Integrated circuits and systems design, page 243, Washington, DC, USA, 2000. IEEE Computer Society.

[Leve. 03] R. Leveugle, L. Antoni, "Dependability Analysis: A New Application for Run-Time Reconfiguration", Int. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, pp. Avril 2003.

[Leve. 09] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence" Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE "09. vol., no., pp.502-506, 20-24 April 2009.

[Leve. 10] R. Leveugle, M. Ben Jrad, "A new methodology for accurate predictive robustness analysis of designs implemented in SRAM-based FPGAs", IEEE International Conference on Electronics, Circuits and Systems (ICECS), Athens, Greece, December 12-15, 2010, pp. 1179-1182.

[Leve. 99] R. Leveugle, "Towards modeling for dependability of complex integrated circuits", 5th IEEE Int. On-Line Testing Workshop (IOLTW "99), Rhodes, Grece, pp. 194-198, Juillet 1999.

[Lewis. 05] D. Lewis, V. Pouget, F. Beaudoin, G. Haller, P. Perdu, P. Fouillat, " Implementing laser-based failure analysis methodologies using test vehicles', IEEE Trans. on Semiconductor Manufacturing, vol. 18, no2, pp. 279–288, Mai 2005.

[Madei. 94] H. Madeira, M. Rela, F. Moreira, J.G. Silva, " RIFLE : A general purpose pin-level fault injector ", Proc. First European Dependable Computing Conference, pp. 199–216, 1994.

[Maist. 13] P. Maistri, S. Tiran, P. Maurine, I. Koren, R. Leveugle, "Countermeasures against EM analysis for a secured FPGA-based AES implementation", International Conference on ReConFigurable Computing and FPGAs (ReConFig' 13), Cancun, Mexico, December 9-11, 2013.

[Mibe. 01] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite", IWWC, 2001

[Miche. 94] T. Michel, R. Leveugle, G. Saucier, R. Doucet, P. Chapier, " Taking advantage of ASICs to improve dependability with very low overheads', Proc. European Design and Test Conference, pp. 14–18, 1994.

[Montes. 07] P. Montesinos, W. Liu, and J. Torrellas. Using register lifetime predictions to protect register files against soft errors. In Dependable Systems and Networks, pages 286–296, 2007.

[MT. 98] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". ACM Trans. on Modeling and Computer Simulation. Vol. 8, no. 1, pp. 3-30, 1998.

[Mukher. 03] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", 36th IEEE International Symposium on Microarchitecture (MICRO-36), 2003, pp. 29-40

[Norm. 94] E. Normand, D. L. Oberg, J. D. Ness, P. P. Majewski, S. Wender, A. Gavron, "Single event upset and charge collection measurements using high energy protons and neutrons", IEEE Transactions on Nuclear Science, Vol. 41, No. 6, December 1994.

[Ott. 04] M. Otto, "Fault Attacks and Countermeasures", Thèse de doctorat, Université de Paderborn, décembre 2004.

[Patt. 05] K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, "Application-Based Metrics for Strategic Placement of Detectors", Pacific Rim Dependable Computing (PRDC), 2005

[Quis. 02] J. Quisquater, D. Samyde, "Eddy current for magnetic analysis with active sensor", Proceedings of Esmart 2002, 3rd ed, September 2002.

[Rand. 09] T. Randy et J. Dick, "The state-of-the-art in ic reverse engineering". International Conference on Cryptographic Hardware and Embedded Systems (CHES), pp. 363-381, 2009.

[Restr. 14] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez, E. Chielle, F. Kastensmidt,"Efficient Metric for Register File Criticality in Processor-Based Systems", Test Workshop - LATW, 15th Latin American, March 12-15, 2014

[Rosc. 13] C. Roscian, A. Sarafianos, J. Dutertre, A. Tria, "Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells", IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 89–98, 2013.

[Sai. 98] F. Saigné, "Une nouvelle Approche de la Sélection des Composants de type MOS pour l'Environnement Radiatif Spatial", Thèse de Doctorat, Université Montpellier II, 1998.

[Samps. 97] J.R. Sampson, W. Moreno, F. Falquez, " Validating fault tolerant designs using laser fault injection (LFI) ", Proc. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT "97), Paris, France, pp. 175–183, Octobre 1997.

[Savi. 12] A. Savino, S. Di Carlo, G. Politano, A. Benso, A. Bosio, G. Di Natale, "Statistical reliability estimation of microprocessor-based systems", IEEE Trans. on Computer, vol. 61, no. 11, pp. 1521-1534, November 2012.

[Schm. 07]. J-M. Schmidt, M. Hutter, "Optical and EM Fault- Attacks on CRT-based RSA : Concrete Results", Proceedings of the 15th Austrian Workshop on Microelectronics (Austrochip), Graz, Austria, 2007.

[Schm. 13] J-M. Schmidt, M. Hutter, "The Temperature Side Channel and Heating Fault Attacks", Smart Card Research and Advanced Application Conference (CARDIS), Springer Berlin Heidelberg, T. 8419, Lecture Notes in Computer Science, pp. 219-235, Berlin, Germany 2013.

[Sext. 98] F. W. Sexton et al, "Precursor ion damage and angular dependence of single event gate rupture in thin oxides", IEEE Transactions on Nuclear Science, Vol. 45, No. 6, pp. 2509-2518, December 1998.

[Skoro. 03] S. Skorobogato, R. Anderson, "Optical Fault Induction Attacks", Conference on Cryptographic Hardware and Embedded Systems (CHES), Springer Berlin Heidelberg, T. 2523, Lecture Notes in Computer Science, pp. 2-12, Berlin, Germany, 2003.

[Skoro. 09] S. Skorobogato, "Local heating attacks on Flash memory devices", IEEE International Workshop on Hardware-Oriented Security and Trust (HOST), 2009.

[Slege. 99] T. J. Slegel, R. M. Averill, III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "IBM's S/390 G5 microprocessor design," IEEE Micro, vol. 19, no. 2, pp. 12–23, Mar.–Apr. 1999.

[SPARC. 92] SPARC international inc. The SPARC Architecture Manual. Version 8. 1991, 1992.

[Srihd. 08] V. Sridharan, and D.R. Kaeli, "Quantifying Software Vulnerability," in Proc. Workshop Radiation Effects and Fault Tolerance in Nanometer Tech. WREFT, pp. 323-328, Ischia, Italy, May 2008.

[Sterp. 07] L. Sterpone, M. Violante, "A new partial reconfiguration-based faultinjection system to evaluate SEU effects in SRAM-based FPGAs", IEEE Transactions on Nuclear Science, vol. 54, issue 4, part 2, August 2007, pp. 965-970.

[Tab. 93] A.Taber, E. Normand, "Single event upset in avionics", IEEE Transactions on Nuclear Science, Vol NS-40, No. 2, pp. 120-126, 1993.

[Trich. 10] E. Trichina, R. Korkikyan, "Multi Fault Laser Attacks on Protected CRT-RSA", IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 75–86, August 2010.

[Vanh. 08] P. Vanhauwaert, "Analyse de Surete par Injection de Fautes dans un Environnement de Prototypage a base de FPGA", Doctorat de l'Institut National Polytechnique de Grenoble (INPG), 2008.

[Vial. 98] C. Vial, "Evaluation de la probabilité des aléas logiques induits par les neutrons atmosphériques dans le silicium des SRAM", Thèse de doctorat, Université Montpellier II, octobre 1998.

[Volk. 97] Volkmar, Sieh, Tschche Oliver et Balbach Frank. 1997. " Comparing Different Fault Models Using VERIFY ". In Proc. 6th Conf. on Dependable Computing for critical Applications. (Grainau, Germany, Mars), p. 59-76.

[Weav. 04] C.T. Weaver, J. Emer, S.S. Mukherjee and S.K. Reinhardt. Reducing the soft-error rate of a high-performance microprocessor. Micro, IEEE, vol. 24, no. 6, pp. 30–37, 2004.

[WW. 1] http://www.nascom.nasa.gov/home.html.

[WW. 2] http://www.xilinx.com/univ/xupv5-lx110t-bsb.htm.

[WW. 3] Laser faut simulator : free software, http://www.anr-liesse.fr/index.php/fr/laser-faut-simulator-free-software.

[WW. 4] Laser fault emulator : free fault injection platform on Virtex5 boards, http://users-tima.imag.fr/amfors/leveugle/ATE-FIT5 Page/ATE-FIT5.htm.

[Zuss. 12] L. Zussa, J. Dutertre, J. Clediere, B. Robisson, A. Tria, "Investigation of timing constraints violation as a fault injection means", 27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France, 2012.

Publications de l'auteur

Conférences internationales :

K. Chibani, M. Portolan, R. Leveugle, "Fast register criticality evaluation in a SPARC microprocessor", 10th Conference on Ph.D Research in Microelectronics and Electronics (PRIME "14), pp. 1-4, Grenoble, FRANCE, 30 June - 3 July, 2014.

K. Chibani, M. Ben Jrad, M. Portolan, R. Leveugle, "Fast accurate evaluation of register lifetime and criticality in a pipelined microprocessor", 22nd IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'14), pp. 260-265, Playa del Carmen, MEXICO, October 5-8, 2014.

K. Chibani, S. Bergaoui, M. Portolan, R. Leveugle, "Criticality evaluation of embedded software running on a pipelined microprocessor and impact of compilation options", IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 778-781, Marseille, FRANCE, December 7-10, 2014.

K. Chibani, M. Portolan, R. Leveugle, "Evaluating application-aware soft error effects in digital circuits without fault injections or probabilistic computations", 22nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Sant Feliu de Guixols, Catalunya, Spain, July 4-6, 2016.

K. Chibani, M. Portolan, R. Leveugle, "Application-aware soft error sensitivity evaluation without fault injections - Application to Leon3", European Conference on Radiation and its Effects on Components and Systems (RADECS), Bremen, Germany, September 19-23, 2016.

Conférences nationales :

K. Chibani, M. Portolan, R. Leveugle, "Analyse de criticité des registres dans un microprocesseur SPARC", 17èmes Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM "14), pp. 4, Lille, FRANCE, 26 au 28 mai 2014.

Glossaire

A

ACE	Architecturally Correct Execution
AES	Advanced Encryption Standard
AVF	Architecture Vulnerability Factor

B

Bascule	Elément mémoire actif sur front d'horloge
Bit-flip	Inversion de la valeur mémorisée dans une cellule mémoire

C

CLB	Conbinantial Logic Block
CTR	Compile-Time Reconfiguration

F

FIT	Failure In Time
FPGA	Field Programmable Gate Array

L

LEON3	Processeur basé sur l'architecture SPARC v8
LUT	Look-Up-Table

M

MBF	Multiple Bit Flip
MBU	Multiple Bit Upset
MCU	Multiple Cell Upset

N

Netlist	Description de circuit a l'aide des éléments qui le constituent et des interconnexions entre ces éléments

P

Pipeline	Architecture qui découpe l'exécution d'une tache en plusieurs étages qui sont opéré en parallèle, chacun sur une instruction différente.

R

Registre	Groupement de bascules
RTL	Register Transfer Level
RTR	Run-Time Reconfiguration

S

SEE	Single Event Effects
SER	Soft Error Rate
SET	Single Event Transient
SEU	Single Event Upset
SFI	Statistical Fault Injection
Soft Error	Erreur transitoire caractérisée par une modification de donnée réversible ou un état erroné temporaire
SPARC	Scalable Processor ARChitecture
SRAM	Static Random Access Memory

T

TMR	Triple Modular Redundancy

Annexes

A.1. Architecture SPARC v8

A.1.1. Formats des instructions

Sparc étant une architecture RISC, toutes les instructions font la taille d'un mot, c'est à-dire 32 bits. Les instructions Sparc 32 possèdent 3 formats d'instructions (figure A-1).

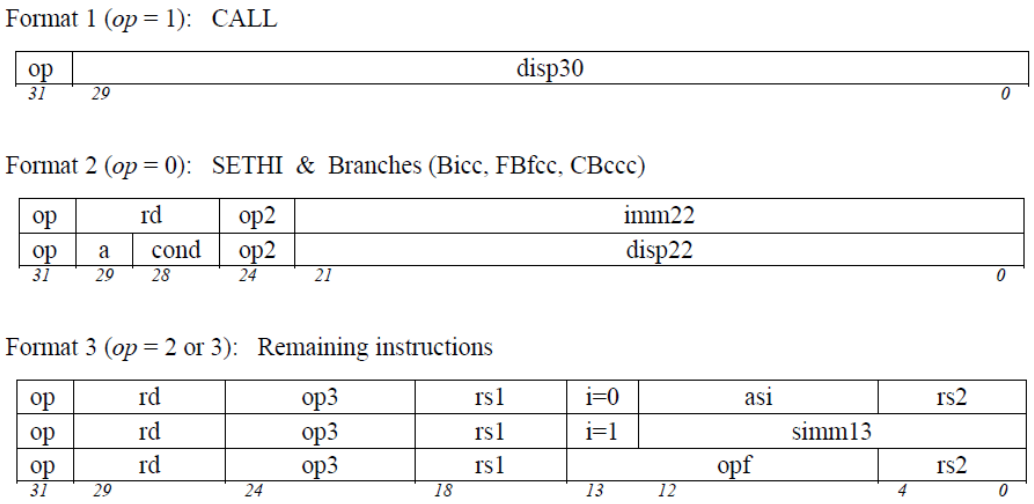


Figure A-1 : Format des instructions dans l'architecture SPARC V8 [SPARC. 92].

Le champ "op" permettant de déterminer le format utilisé :

- **Format** 1 : ce format d'instruction ne sert que pour l'instruction call,
- **Format 2 :** ce format d'instruction sert aux instructions de branchement, ainsi qu'à l'instruction Sethi. Le champ "op2" détermine l'instruction, le champ "rd" contient le numéro du registre de destination, Le champ "imm22" contient une valeur immédiate de 22 bits, le champ "a" contient le bit d'annulation, le champ "cond" contient les codes de condition, et le champ "disp22" contient une valeur immédiate signée,
- **Format 3 :** ce format sert aux instructions load/store, les instructions arithmétiques et logiques, ainsi que les autres instructions. Le champ "op3" détermine l'instruction, le champ "rd" contient le numéro du registre de destination (la plupart du temps), le champ "i" contient un bit spécifiant si le paramètre est immédiat ou non. Les champs "rs1" et "rs2" contiennent les numéros des registres sources. Les champs "asi" et "opf" contiennent respectivement un identificateur d'espace alternatif et un codage d'instruction *floating-point* ou coprocesseur, et ne seront pas utilisés dans ce travail.

L'utilisation de tous ces champs dépend évidemment de l'instruction en faisant usage, et leur sémantique peut dévier en fonction de l'instruction (par exemple l'instruction st (store) utilise rd comme numéro de registre source).

Certains champs utilisés pour des valeurs immédiates ont une taille limitée, car le modèle RISC n'autorise pas les instructions à taille variable.

A.1.2. Instructions

Il y a 72 instructions de base [SPARC. 92]. Les instructions sont toutes encodées sur 32 bits et sont classées en six catégories :

- **Instruction Load / Store**

Les instructions load/store sont les seules ayant accès à la mémoire. Elles utilisent, soit les valeurs de deux registres soit la valeur d'un registre et une valeur immédiate, pour calculer une adresse mémoire sur 32 bits. Cette adresse peut référencer un octet (8 bits), un demi-mot (16 bits) ou un mot (32 bits).

- **Instructions sur nombres entier**

Les instructions sur nombres entiers ont généralement trois opérandes (deux sources et une destination). Les deux opérandes sources sont soit deux valeurs contenues dans des registres soit une valeur contenue dans un registre et une valeur immédiate. La plupart de ces instructions sont disponibles en deux versions. L'une de ces versions positionne le champ icc (*integer condition codes*) du registre PSR (*Processor State Register*), l'autre non. Parmi ce type d'instruction se trouve des instructions pour les additions/soustractions, les multiplications/divisions, les instructions de décalage ainsi qu'une instruction permettant de positionner les bits de poids forts d'une adresse.

- **Instructions de lecture / écriture des registres d'état**

Les instructions de lecture/écriture des registres d'état du processeur permettent d'accéder aux valeurs de ces derniers. Elles permettent ainsi de copier dans (et écrire depuis) un registre général la valeur d'un registre d'état.

- **Instructions de contrôle**

Une instruction de contrôle change la valeur future du compteur programme (nPC). Il y a 7 instructions différentes :

  - Bicc : branchement conditionnel (en fonction de la valeur de icc) ou inconditionnel,
  - CALL et JMPL : sauvegarde de PC puis saut à l'adresse calculée,
  - Ticc : génération d'une trappe inconditionnellement ou en fonction du résultat d'un test sur la valeur de icc,

- RETT : retour d'une trappe ou d'une fonction,
- SAVE et RESTORE : glissement de la fenêtre de registre en incrémentant ou décrémentant la valeur de cwp (instructions utilisées en début et en fin de fonction).

- **Instructions flottantes**

Ces instructions ne sont pas implémentées dans la version du LEON3 utilisée, ce sont les instructions faisant appel à l'unité de calculs flottants.

- **Instructions pour le coprocesseur**

Ces instructions ne sont pas implémentées dans la version du LEON3 utilisée, ce sont les instructions faisant appelle au coprocesseur.

A.1.3. Banc de registres et manipulation des fenêtres

Certains des registres généraux ont des caractéristiques particulières, le tableau A-1 les résume.

Tableau A-1 : Les registres SPARC v8.

	Registres	Numéro	Description
Global	%g0	0	Toujours égal à 0
	%g1	1	Valeur temporaire
	%g2 -%g4	2-4	Registres globaux
	%g5-%g7	5-7	Réservés pour Sparc ABI
Out	%o0	8	Paramètre sortant (0) / Valeur de retour de la fonction appelée
	%o1-o5	9-13	Paramètres sortants
	%o6	14	Pointeur sur la pile %sp
	%o7	15	Valeur temporaire / adresse de l'instruction CALL
Local	%l0 -%l7	16-23	Registres locaux
In	%i0	24	Paramètre entrant (0) / Valeur de retour de la fonction appelante
	%i1-i5	25-29	Paramètres entrants
	%i6	30	Pointeur sur la fenêtre de registres %fp
	%i7	31	Adresse de retour

A.1.3.1. Manipulation des fenêtres

Pour changer de fenêtre, nous avons deux appels assembleur :

- L'appel SAVE : cet appel est fait avant d'appeler une fonction. Dans ce cas :
    - Les registres globaux ne changent pas,
    - Le pointeur de fenêtre se décrémente,
    - Les registres de sortie de la fenêtre courante deviennent les registres d'entrée de la nouvelle fenêtre,
    - Le processeur alloue de nouveaux registres locaux et sortie (prend ceux de la fenêtre suivante).
- L'appel RESTORE : cet appel est fait une fois que la fonction est terminée. Dans ce cas :
    - Les registres globaux ne changent pas.
    - Le pointeur de fenêtre s'incrémente.
    - Les registres d'entrée de la fenêtre courante deviennent les registres de sortie de la nouvelle fenêtre.
    - Le processeur retrouve les registres locaux et sortie de la fonction à laquelle on est retourné.

A.1.3.2. Cas extrêmes dans la manipulation des fenêtres : Overflow et Underflow

Même si le nombre de fenêtres est élevé, donc même si on peut appeler, rappeler et encore appeler des fonctions les unes à la suite des autres, il y a un moment où une fonction fera un SAVE et qu'il n'y aura plus de fenêtres disponibles. Ceci est appelé un "*overflow*" et génère une exception (SPILL) qui est gérée par le système d'exploitation.

Le système d'exploitation est alors chargé de sauvegarder des fenêtres (quelques-unes ou toutes) dans un endroit (probablement une pile) et les marquer comme étant disponibles. Le processeur peut alors satisfaire l'instruction SAVE et la fonction continue à s'exécuter.

Similairement, quand une fonction fait un RESTORE alors que la fenêtre à restaurer n'est plus disponible (est dans la pile), ceci génère l'exception "*underflow*" (FILL). Le système d'exploitation est alors chargé de restaurer des fenêtres (quelques-unes ou toutes) et les marquer comme étant disponibles. Le registre WIM (*Window Invalid Mask*) permet de savoir si les fenêtres sont déjà toutes utilisées. Ce registre n'est pas utilisé par les instructions mais par le matériel.

A.2. Effets des caractéristiques architecturales sur la criticité

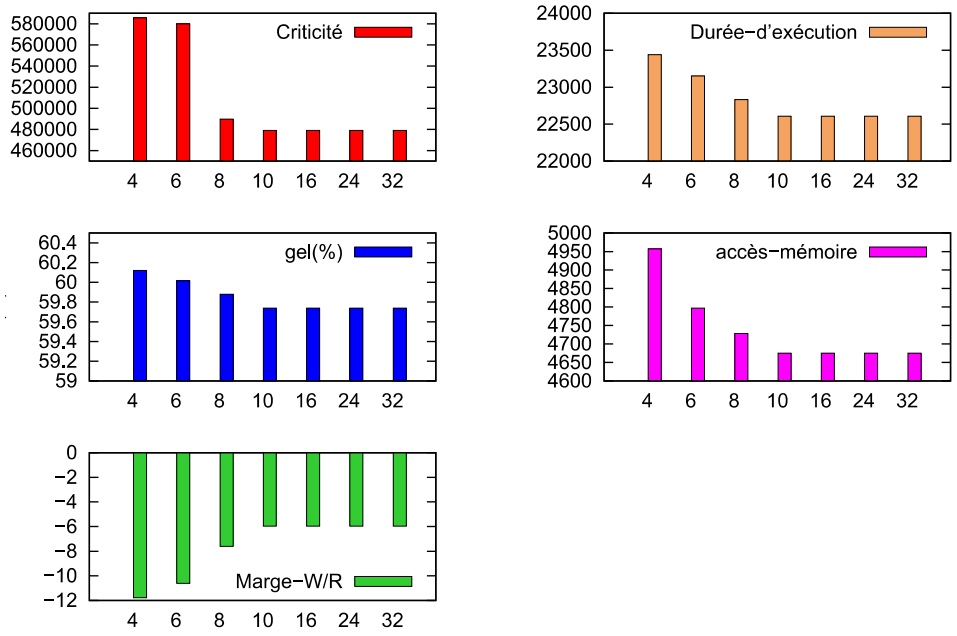


Figure A-2 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application AES.

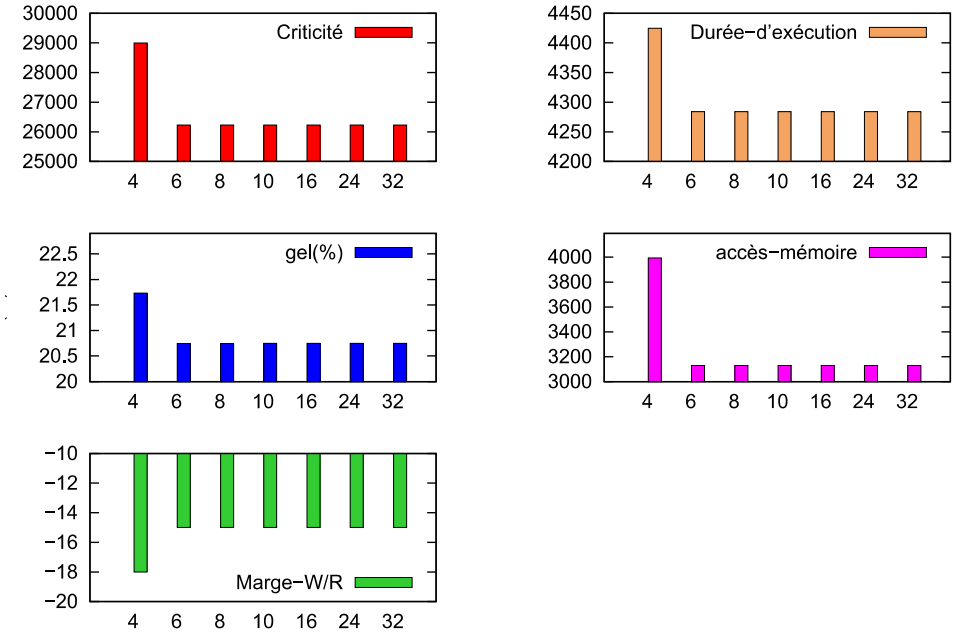


Figure A-3 : Effets de la variation du nombre de fenêtres dans le banc de registres, pour l'application FIR.

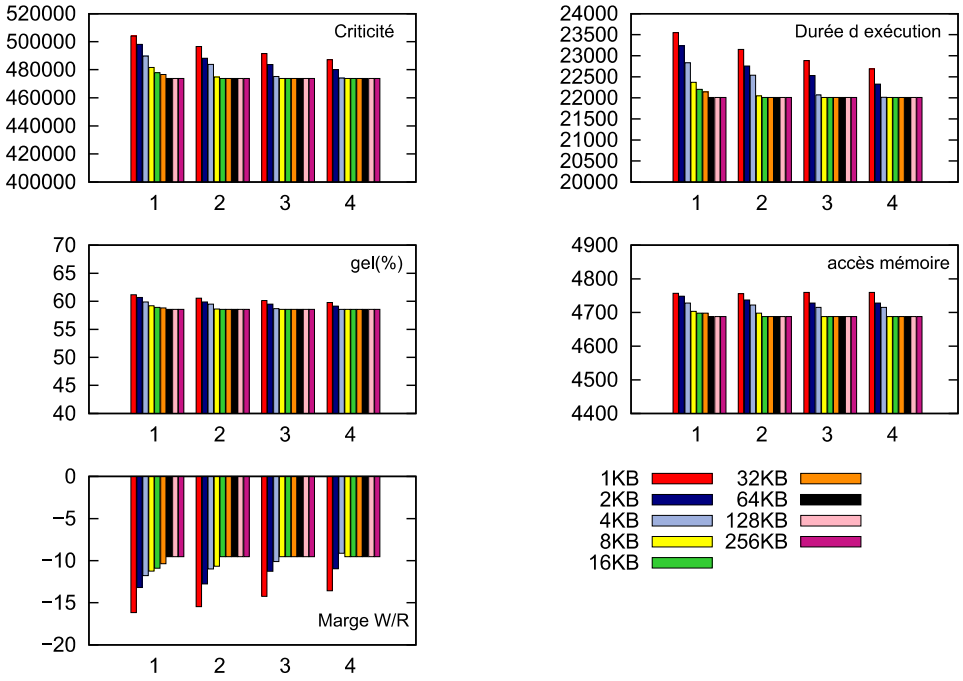


Figure A-4 : Effets de la variation du nombre de sous-ensembles et de la taille des caches, pour l'application AES.

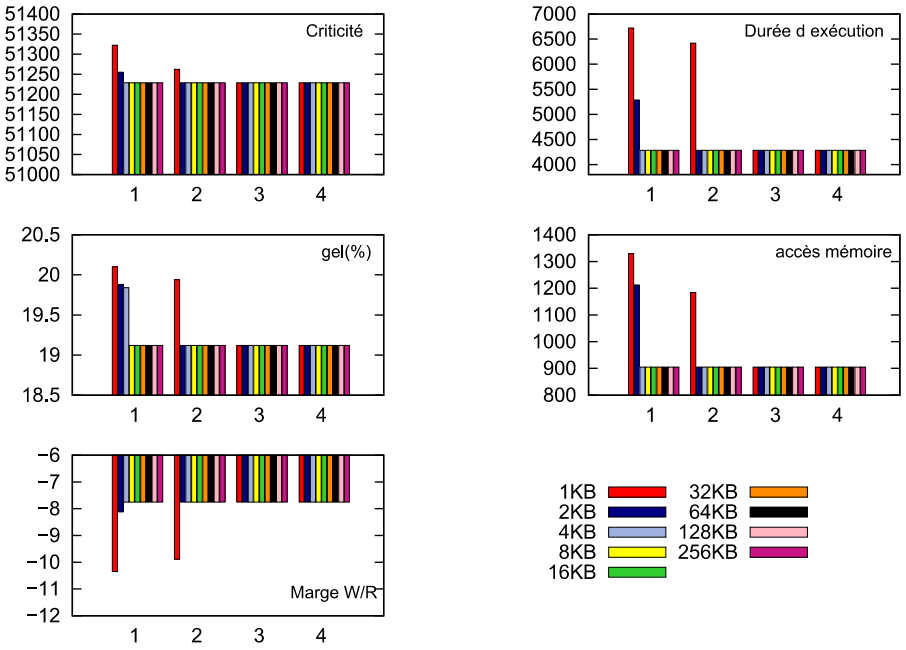


Figure A-5 : Effets de la variation du nombre de sous-ensembles et de la taille des caches, pour l'application FIR.

TITRE : Analyse de robustesse de systèmes intégrés numériques.

RESUME : Les circuits intégrés ne sont pas à l'abri d'interférences naturelles ou malveillantes qui peuvent provoquer des fautes transitoires conduisant à des erreurs (*Soft errors*) et potentiellement à un comportement erroné. Ceci doit être maîtrisé surtout dans le cas des systèmes critiques qui imposent des contraintes de sûreté et/ou de sécurité. Pour optimiser les stratégies de protection de tels systèmes, il est fondamental d'identifier les éléments les plus critiques. L'évaluation de la criticité de chaque bloc permet de limiter les protections aux blocs les plus sensibles. Cette thèse a pour objectif de proposer des approches permettant d'analyser, tôt dans le flot de conception, la robustesse d'un système numérique. Le critère clé utilisé est la durée de vie des données stockées dans les registres, pour une application donnée. Dans le cas des systèmes à base de microprocesseur, une approche analytique a été développée et validée autour d'un microprocesseur SparcV8 (LEON3). Celle-ci repose sur une nouvelle méthodologie permettant de raffiner les évaluations de criticité des registres. Ensuite, une approche complémentaire et plus générique a été mise en place pour calculer la criticité des différents points mémoires à partir d'une description synthétisable. L'outil mettant en œuvre cette approche a été éprouvé sur des systèmes significatifs tels que des accélérateurs matériels de chiffrement et un système matériel/logiciel basé sur le processeur LEON3. Des campagnes d'injection de fautes ont permis de valider les deux approches proposées dans cette thèse. En outre, ces approches se caractérisent par leur généralité, leur efficacité en termes de précision et de rapidité, ainsi que leur faible coût de mise en œuvre et leur capacité à ré-exploiter les environnements de validation fonctionnelle.

Mots clés : Circuits numériques, Evaluation de robustesse, Systèmes critiques, Fautes transitoires, Criticité de registres, Injection de fautes, LEON3.

TITLE : Robustness analysis of digital integrated systems.

ABSTRACT : Integrated circuits are not immune to natural or malicious interferences that may cause transient faults which lead to errors (soft errors) and potentially to wrong behavior. This must be mastered particularly in the case of critical systems which impose safety and/or security constraints. To optimize protection strategies of such systems, it is essential to identify the most critical elements. The assessment of the criticality of each block allows limiting the protection to the most sensitive blocks. This thesis aims at proposing approaches in order to analyze, early in the design flow, the robustness of a digital system. The key criterion used is the lifetime of data stored in the registers for a given application. In the case of microprocessor-based systems, an analytical approach has been developed and validated on a SparcV8 microprocessor (LEON3). This approach is based on a new methodology to refine assessments of registers criticality. Then a more generic and complementary approach was implemented to compute the criticality of all flip-flops from a synthesizable description. The tool implementing this approach was tested on significant systems such as hardware crypto accelerators and a hardware/software system based on the LEON3 processor. Fault injection campaigns have validated the two approaches proposed in this thesis. In addition, these approaches are characterized by their generality, their efficiency in terms of accuracy and speed and a low-cost implementation. Another benefit is also their ability to re-use the functional verification environments.

Keywords : Digital circuits, Robustness evaluation, Critical systems, Transient faults, Registers criticality, Fault injection, LEON3.

INTITULE ET ADRESSE DU LABORATOIRE

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

ISBN : 978-2-11-129-217-8