Méthodes et outils pour l'analyse tôt dans le flot de conception de la sensibilité aux soft-erreurs des applications et des circuits intégrés

Wassim Mansour

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **« Micro et Nano Électronique»**

Arrêté ministériel : 7 août 2006

Présentée par

Wassim Mansour

Thèse dirigée par **Raoul Velazco**

préparée au sein du **Laboratoire TIMA**
dans **l'École Doctorale d'Électronique, Électrotechnique, Automatique, et Traitement du Signal (EEATS)**

Méthodes et outils pour l'analyse tôt dans le flot de conception de la sensibilité aux soft-erreurs des applications et des circuits intégrés

Thèse soutenue publiquement le **31 octobre 2012**,
devant le jury composé de :

M. Bernard COURTOIS
Directeur de recherche au CNRS, Directeur du laboratoire CMP,Grenoble, Président
Mme. Lirida NAVINER
Professeur à l'INS2I Paris, Rapporteur
M. Luis ENTRENA
Professeur à l'UC3M, Madrid, Rapporteur
M. Dan ALEXANDRESCU
Ingénieur-Docteur a iRoC Tech, Grenoble, Membre
M. Wassim El-Falou
Professeur a l'UL, Tripoli, Liban, Membre
M. Raoul VELAZCO
Directeur de recherche au CNRS, Grenoble, Directeur de thèse

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

Remerciements

C’est en premier lieu à mon directeur de thèse, M. Raoul Velazco, que s’adressent mes remerciements. Son encadrement, son dynamisme, ses encouragements, sa confiance, ses conseils et ses critiques m’ont permis de progresser et mener à bien ce travail. Qu'il trouve ici l'expression de ma sincère reconnaissance.

Je tiens à exprimer ma profonde gratitude pour les membres du jury:

Monsieur Bernard Courtois, pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse.

Madame Lirida Naviner et Monsieur Luis Entrena, pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de ce travail.

J'adresse mes remerciements à Monsieur Dan Alexandrescu et Monsieur Wassim El-Falou pour leur participation à mon jury en temps qu'invités.

Je souhaiterais remercier Madame Dominique Borrione, Directeur de Recherche au CNRS et Directeur du Laboratoire pour m'avoir accueilli au sein de TIMA ainsi que tous les administratifs du laboratoire TIMA, pour leur aide, leur disponibilité et leur humeur.

Je remercie profondément mes profs. Libanais, Monsieur Rafic Ayoubi, Monsieur Haissam Ziade qui, grâce à leur confiance, cette thèse a eu lieu.

Je souhaite remercier Paul Peronnard et Gilles Foucard qui m'ont fait partager leurs connaissances et compétences techniques au début de ma thèse. Je remercie également Fabrice Pancher pour partager les précieuses discussions pour nos recherches.

Je remercie chaleureusement tous mes collègues au laboratoire TIMA, très particulièrement Diarga Fall, Gilles Bizot, Hai Yu, Saif Ur Rehman, Yi Gang, Vladimir Paska, Fabien Chaix, Michael Dimopolous, Greicy Marques-Costa, Adrian Evans, Hamayun Muhammad, Mohamed Ben Jrad et Salma Bergaoui. Une pensée pour les libanais que j'ai rencontré à TIMA durant ces trois ans, surtout à Mariam Abdallah.

Cette thèse n'aura pas être possible sans le support de plusieurs personnes qui m'ont entouré aimé et encouragé. Je remercie tout particulièrement mes amis proches, Hassan (COMA), Georges Z.,

Elias, Ayman, Georges M., mon cousin Hassan pour leur encouragement et leur profonde amitié durant ces trois ans.

Finalement je réserve mon dernier merci, et combien c'est fort, spécial et sincère à ma petite Soumi qui a supporté les moments de bonheur ainsi que les moments les plus difficiles pendant ces trois dernières années, mon frère Hanna, ma famille surtout ma grand-mère Adèle et mes parents Marwan et Laurice pour tout le soutien qu'ils m'ont apporté dans ce travail de longue haleine.

Wassim Mansour

Table des matières

Liste des figures

Liste des tableaux

Liste des abbreviations

AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CE	Chip Enable
CEU	Code Emulated Upset
CLB	Configuration Logic Block
CLK	Clock
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DLL	Dynamic Link Library
DUT	Device Under Test
EPROM	Erasable Programmable Read Only Memory
FD	D Flip-flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HDL	Hardware Description Language
LET	Linear Energy Transfer
LUT	Look-Up Table
LWS	Living With Star
MAC	Media Access Control
MBU	Multiple Bit Upset
MCU	Multiple Cell Upset
MPTB	Micro-electronic and Photonic TestBed
NIEL	Non Ionizing Energy Loss
PC	Program Counter

PCI	Peripheral Component Interconnect
PSOC	Programmable System On Chip
QARM	Qinetiq Atmospheric Radiation Model
RAM	Random Access Memory
RTL	Register Transfer Language
SDRAM	Synchronous Dynamic Random Access Memory
SEB	Single Event Burnout
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEL	Single Event Latchup
SER	Soft-Error Rate
SET	Single Event Transient
SEU	Single Event Upset
SHE	Single event Hard Error
SOC	System On Chip
SP	Stack Pointer
SRAM	Static Random Access Memory
SSA	South Atlantic Anomaly
STRV	Space Technology Research Vehicles
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
VLSI	Very Large Scale Integration
WE	Write Enable

Introduction

La fiabilité des circuits électroniques peut être affectée par les particules énergétiques présentes dans l'environnement dans lequel ils opèrent, surtout dans l'espace et les hautes altitudes de l'atmosphère terrestre. En effet, le rayonnement cosmique et le soleil au travers d'activités permanentes, comme le vent solaire, ou bien d'événements violents et ponctuels, comme les éruptions solaires et les éjections de masse coronale, peuvent produire de particules hautement énergétiques. Ces particules frappent continuellement l'atmosphère terrestre, qui absorbe une grande partie d'entre elles, tandis que le reste rentre en collision avec les molécules atmosphériques en produisant ce qu'on appelle une *douche cosmique*. Ces particules suite à leurs impact dans des zones sensibles des circuits intégrés peuvent provoquer des pannes transitoires ou permanentes. A la fin des années 70s sont présentées pour la première fois les notions des événements singuliers (SEE) [Guenzer 1979] [May 1979] qui entrainent des effets non désirables voire destructifs sur le fonctionnement des composants. Cette problématique doit être de nos jours, de plus en plus considérée, à cause de la miniaturisation des gravures des transistors qui deviennent trop fins et plus proche l'un de l'autre. Différentes techniques de durcissement ont été présentées depuis les années 80s, pour assurer la fiabilité des systèmes électroniques.

Le prix élevé des composants durcis ainsi que la réduction des budgets pour des projets spatiaux, font que la tendance, depuis les années 2000, est à l'utilisation, dans des applications spatiales et atmosphériques [Chao 2000] [DiUbaldo 2000], des composants commerciaux (*Commercial Off-The-Shelf* ou *COTS*), caractérisés par leurs bonnes performances, leurs couts relativement faibles et leurs disponibilités en quantité. Ceci impose la détermination de leurs sensibilités face aux effets des radiations qui peuvent perturber leur fonctionnement. Plusieurs études ont été menées pour mettre en évidence la sensibilité des circuits intégrés lorsqu'ils opèrent en altitude avionique [Normand 1996] et même au niveau du sol. Les expériences réalisées sur des circuits complexes à base de mémoires SRAM (Static Random Access Memory) [Baumann 2005], ont montrés la possibilité de l'occurrence des événements multiples tel les MCU (Multiple Cell Upset) et le MBU (Multiple Bit Upset).

L'évaluation de la sensibilité des circuits intégrés face aux effets des particules énergétiques devient de nos jours un aspect crucial afin de prédire le comportement du circuit cible dans son environnement final et appliquer si nécessaire des techniques appropriées de tolérance aux fautes. Parmi les méthodologies développées dans ce but on peut distinguer deux catégories : les essais en

environnement réel (Real-Life Test) et les tests accélérés au sol (Accelerated Radiation Ground Tests). Les essais en environnement réel consistent à exposer le circuit cible aux radiations atmosphériques lors des vols longs courriers [Duzzelier 1997], dans des ballons stratosphériques [Taber 1993] et dans des villes et des montagnes à hautes altitudes [Lesea 2005]. Les tests accélérés ont pour but d'évaluer la sensibilité face aux effets des radiations en exposant le circuit cible à des faisceaux de particules énergétiques lorsqu'il exécute une application. Les sources radioactives, les accélérateurs de particules et les faisceaux lasers sont les moyens de test utilisés pour ces expériences. Dans le but d'étudier le comportement d'un circuit cible face aux particules énergétiques, diverses méthodes d'injection de fautes (simulation/emulation) sont utilisées [Sieh 1997] [Folkesson 1998] [Rahbaran 2004] [Shokrallah 2008]. Ces méthodes peuvent être divisées en deux catégories principales : Injection de fautes basée sur le matériel ou HWIFI et Injection de fautes à base logiciel: Ces méthodes peuvent être divisées en deux classes, les logiciels qui implémentent l'injection des fautes ou SWIFI (Software-Implemented Fault-Injection), et les injections de fautes basées sur la simulation.

Cette thèse a pour objectif l'étude d'une méthode générale et automatisable pour la prédiction de la sensibilité des circuits intégrés, face aux radiations présentes dans l'environnement naturel dans lequel ils vont opérer. La méthode proposée se base sur l'émulation des fautes de type SEU (basculement des bits dans une cellule mémoire) et SET (génération d'un pulse de courant dans un circuit combinatoire), dans des circuits cibles implémentées sur un FPGA. Ceci suppose que le modèle RTL est disponible. L'idée est de synthétiser un code RTL pour la génération de son netlist, qui sera modifiée pour permettre l'injection des fautes considérées dans la zone sensible, cellule mémoire, entrée/sortie d'une porte logique, aléatoirement choisie parmi les ressources sensibles du circuit étudié. L'instant d'injection de fautes est également choisi aléatoirement. La contribution majeure de cette méthode est la possibilité d'injecter les fautes considérées et des fautes de type Stuck_at, pour identifier un défaut de fabrication dans un circuit, dans toutes sortes de ressources, incluant pour ce qui est de cellules mémoires celles des bascules de la partie contrôle, des registres, mémoire de toutes sortes, ceci sans aucune restriction sur la complexité du circuit. Cette méthode est aussi automatisable, et a besoin d'une configuration minimale du FPGA sur lequel elle sera appliquée.

Dans le premier chapitre, seront présentés les sources de rayonnement ainsi que les environnements radiatifs spatial et atmosphérique. Les interactions de particules énergétiques avec les circuits intégrés seront ensuite décrites, afin d'aborder les différents effets qui peuvent avoir lieu lors de cette interaction. Les résultats des tests effectués durant cette thèse en exposant des circuits, de type mémoires SRAM, à l'effet de l'environnement réel seront présentés pour mettre en évidence la pertinence de cette thématique pour des circuits et des applications modernes.

Les méthodes et les outils de test permettant la caractérisation des circuits intégrés face aux radiations sont décrits dans le chapitre 2. La plateforme de test ASTERICS ainsi que la méthode

d'injection de fautes CEU (*Code Emulated Upset*), développées à TIMA dans le cadre des recherches précédentes, seront utilisées pour l'injection de fautes de types SEU, dans un microcontrôleur PSOC exécutant une application benchmark, ceci pour mettre en évidence le besoin d'une méthode plus générale.

La méthode d'injection de fautes proposée pour faire face à cette conjoncture, méthode qui fut appelée NETFI (NETlist Fault Injection), sera décrite dans le chapitre 3. Dans le but de la validation de la nouvelle stratégie, la méthode sera appliquée à différents types de circuits. La première cible sera un processeur complexe: le LEON2. Une deuxième cible, sera un microcontrôleur 80C51 d'Intel dont les résultats issus des tests en accélérateurs de particules sont disponibles dans [Rezgui 2001]. Une comparaison des prédictions issues de la méthode NETFI avec les résultats issus des tests sous radiations sera présentée pour valider l'efficacité de la méthode proposée. Un dernier circuit cible sera un Réseau de Neurones Artificiels (RNA) de type Hopfield (RNH) conçu en deux versions : une originale et une tolérante aux fautes via la redondance matérielle et logicielle. Les résultats des expériences d'injection de fautes faites avec NETFI sur ces deux versions du RNH seront présentés pour fournir des données sur la robustesse de la méthode tolérante aux fautes.

Dans le chapitre 4, l'injection de fautes sera réalisée sur un processeur LEON3 lorsqu'il exécute un algorithme tolérant aux fautes, dit *self-convergent*. Ces types d'algorithmes, considérés immune aux fautes, sont utilisés dans la communication entre les nœuds des réseaux (de processeurs, de capteurs, etc.). Les résultats de l'application des méthodes d'injection de fautes, CEU et NETFI, seront comparés pour mettre en évidence la robustesse et les potentielles tallons d'Achilles de cet algorithme et les solutions qui peuvent être utilisées pour y faire face.

Dans le dernier chapitre seront présentées les conclusions ainsi que les futures perspectives de ces travaux.

Chapitre 1. Les radiations et leurs effets sur les circuits intégrés

Ce chapitre introduit brièvement la thématique des radiations en général présentant un aperçu sur la composition des couches du soleil ainsi que sur les environnements radiatifs naturels, spatial et atmosphérique, et leurs effets sur les circuits intégrés. Des données issues de tests à hautes altitudes sont décrites pour clairement mettre en évidence la criticité de cette problématique.

1.1 Historique

Il est bien connu que notre système planétaire est composé d'une étoile et des corps gravitant autour d'elle. Cette étoile ou soleil, située au sein de notre galaxie, constitue avec les rayons cosmiques l'origine de toutes les radiations. Chaque seconde son cœur fusionne environ 700 millions de tonnes d'hydrogène et produit 695 millions de tonnes d'hélium et 5 millions de tonnes d'énergie sous forme de rayons gamma [Eddy 2009]. Ceci explique l'énergie dégagée par le soleil qui est d'environ 386 milliards de milliards de mégawatts.

La surface du soleil est appelée la **photosphère** et a une température d'environ 5500°C. Au dessus de la photosphère repose une petite couche appelée la **chromosphère**. La région pauvre en éléments, au dessus de la chromosphère s'appelle la **couronne**, et s'étend à des millions de kilomètres dans l'espace. On peut l'observer lors d'une éclipse seulement, et sa température peut dépasser un million de degrés Celsius. Le champ magnétique du soleil est très puissant par rapport au standard terrestre. Sa **magnétosphère** appelée aussi **héliosphère** s'étend bien plus loin que Pluton [Eddy 2009].

Plusieurs études ont été faites sur les radiations ainsi que sur leurs conséquences sur tout ce que les rencontre ou entre en réaction avec, surtout les circuits intégrés. Ces études ont été commencées depuis la fin des années 70s et montrent que les composants électroniques peuvent être perturbés par les radiations de l'environnement dans lequel ils opèrent. En 1975, Binder présente la première publication sur le rôle probable des radiations sur les circuits imprimés placés dans un satellite [Binder 1975]. En 1979 les recherches faites par Ziegler et Lanford trouvent que les radiations jouent un rôle très important en altitudes avioniques, et que les perturbations qui viennent des réactions neutron-silicium doivent être considérées pour les circuits et systèmes destinés à opérer à hautes altitudes [Zeigler 1996].

Le développement de la technologie VLSI (Very-Large-Scale Integration) rend plus importante la probabilité d'apparition des perturbations dues aux radiations. Ce sujet doit donc être pris en compte pour assurer le bon fonctionnement des systèmes n'importe où ils évoluent, dans l'espace, dans l'atmosphère et même pour celles opérant au sol.

1.2 Environnement radiatif

Cette section a pour but de présenter l'environnement radiatif, spatial et atmosphérique, dans lequel opèrent les composants électroniques.

1.2.1 Environnement radiatif spatial

Dans l'environnement spatial, les perturbations des circuits intégrés sont extrêmement dépendantes des caractéristiques des rayonnements incidents (énergie et flux) ainsi que de leur probabilité d'apparition. Ils sont généralement soumis à l'effet des électrons, des protons et des ions, d'origines et d'énergies diverses [Boudenot 1995].

L'environnement radiatif spatial se compose principalement de trois sources radiatives suivant leurs origines: le rayonnement cosmique (Global Cosmic Rays ou GCR), le vent et les éruptions solaires et les ceintures de radiations [Stapor 1988] [Tang 2004].

- Le rayonnement cosmique

Jusqu'à nos jours, l'origine principale de ce rayonnement reste inconnue, on sait seulement qu'il s'agit d'un rayonnement provenant de sources galactiques et extragalactiques, d'après les résultats des recherches découvertes par V. Hess en 1912 [Bourrieau 1991]. Il est constitué de protons (87%), d'hélium (12%) et d'ions lourds (1%) ayant de très grandes énergies (supérieure à 1 GeV et jusqu'à 10^{11} Gev) [Fleischer 1975]. Dans le système solaire, le flux qui caractérise la population des ions cosmiques est modulé par le cycle de l'activité solaire: le vent solaire s'oppose au flux des rayons cosmiques quand le soleil est en période d'activité maximale. Les rayonnements sont donc très énergétiques mais les flux associés sont relativement faibles. Il est nécessaire de les prendre en compte dans le cas des missions spatiales longues de plusieurs années car la probabilité d'apparition d'un évènement potentiellement destructif n'est pas négligeable. Dans la figure 1.1 est montré le flux du rayonnement cosmique en fonction de l'énergie des particules, alors que dans la figure 1.2 sont donnés les flux relatifs en fonction du type d'ion.

- Le vent et les éruptions solaires

Le soleil est une étoile qui émet sa propre lumière dont son énergie est d'origine nucléaire [Eddington 1926]. Il représente 99,8% de la masse totale du système solaire tandis que tout le reste représente seulement 0,2% incluant la terre. Il est principalement constitué d'hydrogène (90%) et d'hélium (8%). Les réactions thermonucléaires de fusion qui ont lieu au centre du soleil peuvent convertir l'hydrogène en hélium. L'énergie produite à cette occasion est émise sous forme de lumière, de radiations et de particules.

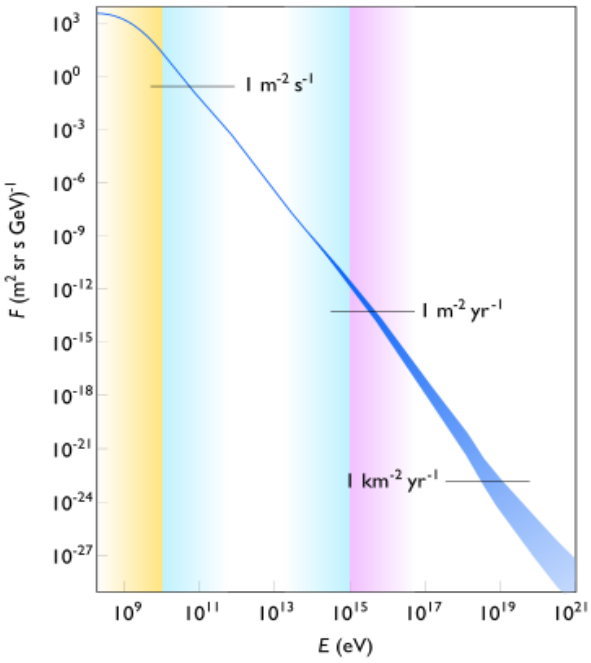


Figure 1.1 Flux du rayonnement cosmique en fonction de l'énergie de particules

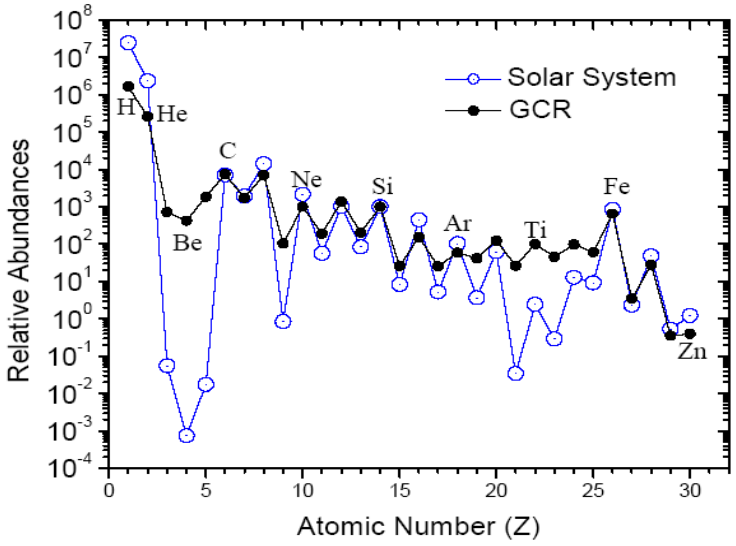


Figure 1.2 Flux relatif des différents éléments constituant les rayons cosmiques

Deux types d'activités influencent l'environnement radiatif. La première est continue, c'est le vent solaire tandis que la seconde est périodique, ce sont les éruptions solaires [Boudenot 1995].

Le vent solaire est un flot de particules chargées s'échappant continuellement de la haute atmosphère (ou couronne) du soleil situé au-delà de la chromosphère. Il est essentiellement constitué d'électrons, de protons et d'atomes d'hélium avec des traces infimes d'ions d'éléments plus lourds tels

l'oxygène ou le carbone. Il s'échappe en permanence et dans toutes les directions de la surface du soleil et baigne l'ensemble du système solaire emportant avec lui les lignes de champ magnétique.

La figure 1.3 représente le soleil (au milieu) avec tout autour l'atmosphère solaire ou couronne. Le graphique montre la vitesse du vent solaire mesurée par le satellite Ulysse : le vent solaire rapide est expulsé des régions autour des pôles du soleil à la vitesse de 800 Km/s, alors que le vent solaire lent écoule de la zone équatoriale à 350 Km/s. Le vent solaire est immergé dans un champ magnétique. Le graphique est coloré en rose quand la polarité du champ magnétique solaire (c.à.d. la direction qu'indiquerait l'aiguille d'une boussole) pointe dans la direction opposée au soleil (au cours du cycle solaire actuel, il s'agit de la direction du champ magnétique au-dessus du pôle Nord du soleil) et coloré en bleu quand la polarité du champ magnétique solaire pointe vers le soleil (la direction du champ magnétique au-dessus du pôle Sud).

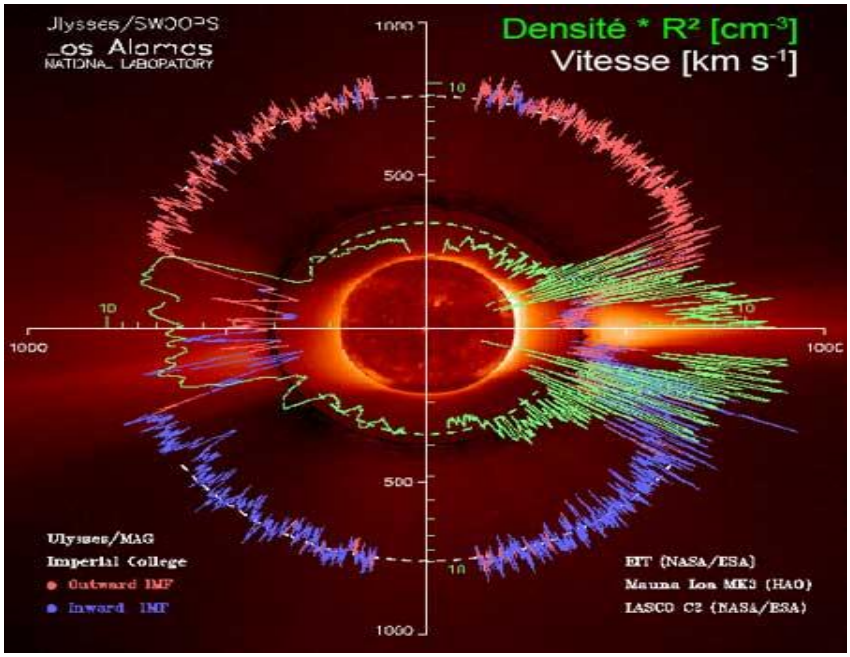


Figure 1.3 : Mesure du vent solaire fournie par le satellite Ulysse

Une éruption solaire est un événement primaire de l'activité du soleil. Elle se produit à la surface de la photosphère et projette au travers de la chromosphère un jet de matière ionisée qui se perd dans la couronne à des centaines de milliers de Km d'altitude. En plus, l'éruption s'accompagne d'un intense rayonnement qui provoque l'apparition des aurores polaires en entrant en interaction avec le champ magnétique terrestre et qui peut perturber les transmissions radioélectriques terrestres.

L'activité solaire se matérialise par la présence des taches à la surface de la photosphère. L'observation de ces taches n'est pas récente, Galilée les avait mis en évidence en 1610. Dans la

figure 1.4 fournie par la NASA, prise dimanche 22 janvier 2012, est montrée une éruption dans l'hémisphère Nord du soleil.

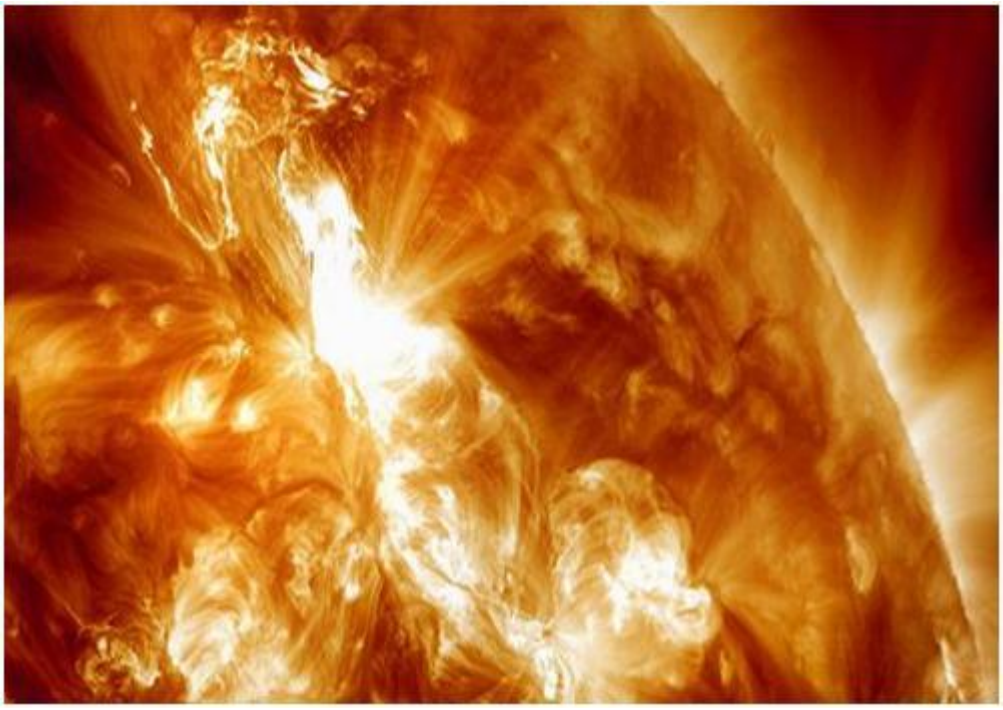


Figure 1.4 : Les éruptions dans l'hémisphère Nord du soleil en janvier 2012

Les taches ne sont pas uniformément réparties dans la surface du soleil: elles se limitent généralement aux zones de 30 degrés au Nord et au Sud de l'équateur solaire. La figure 1.5 montre la répartition des taches dans la surface du soleil. Ces taches se composent d'une partie centrale (l'ombre), qui est moins lumineuse que le reste de la photosphère (fine couche de 300 Km qui rayonne la lumière visible) car elle est moins dense et plus froide (4200°K) que l'atmosphère normale (5800°K), et d'une pénombre (région de transition entourant l'ombre) dont la température n'est inférieure que de 300 à 500 °K à celle de la photosphère. Le diamètre des taches peut atteindre 120000 Km, et si elles sont regroupées, elles peuvent s'étendre de l'Est vers l'Ouest sur 250000 Km.

L'activité solaire n'est pas constante au cours du temps. Comme l'indique le diagramme de Maunder (figure 1.6), on observe la présence d'un grand nombre de centres actifs durant des périodes qui se répètent tous les 11 ans en moyenne. Mais, des mesures plus poussées laissent suggérer l'existence d'un second cycle de 80 à 100 ans. Toutefois, il est nécessaire de considérer que le cycle solaire est constitué de deux périodes de 11 ans, soit 22 ans. La figure 1.6 montre le nombre annuel moyen des taches entre les années 1700 et 2000.

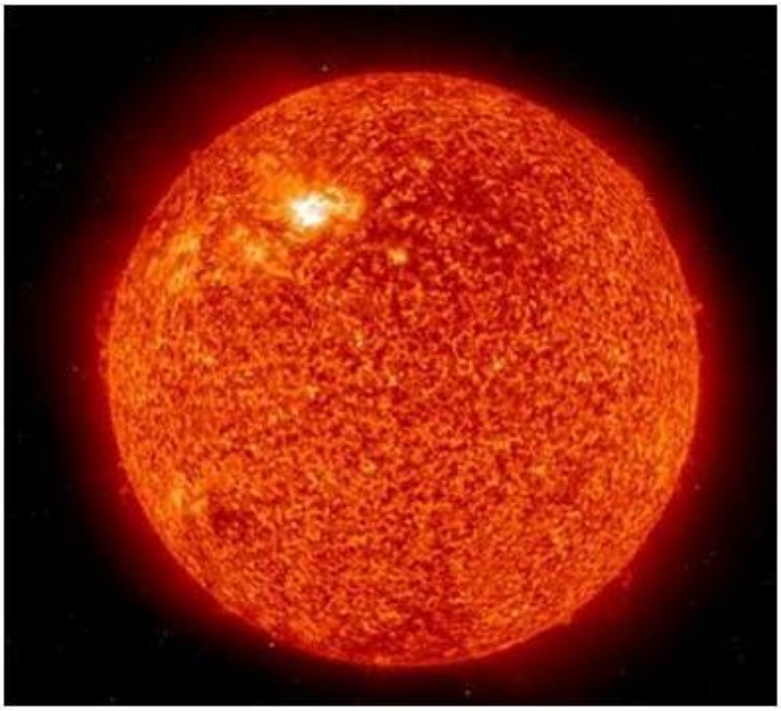


Figure 1.5 : Répartition des taches solaires.

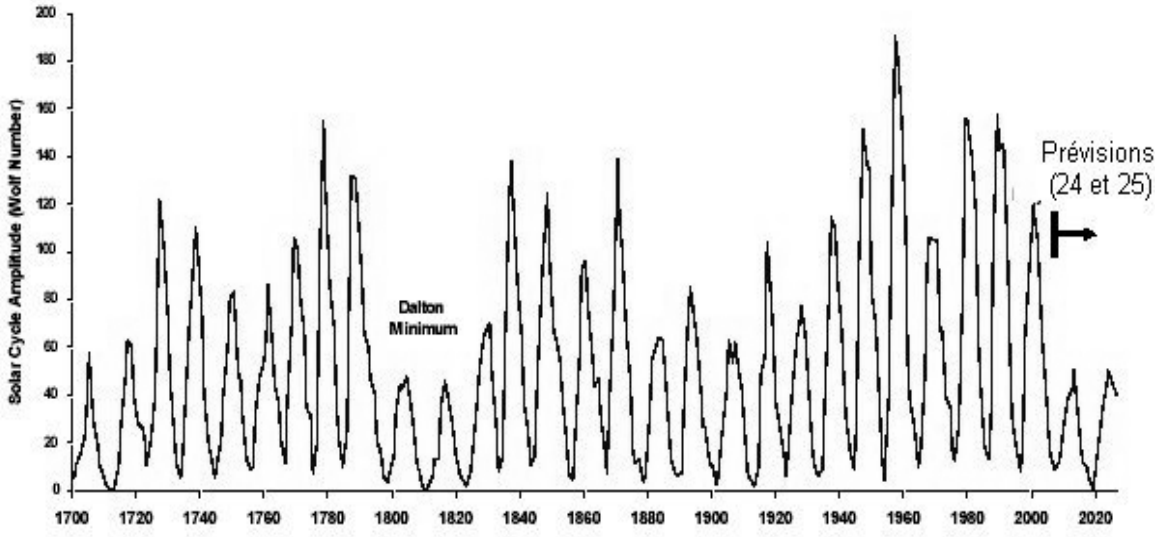


Figure 1.6 : Nombre annuel moyen des taches 1700-2030.

Les éruptions solaires se produisent en période d'activité maximale. Leur apparition est aléatoire et s'accompagne d'une émission des particules: des protons, des électrons et des ions lourds. Les protons peuvent atteindre des énergies de l'ordre de centaines de MeV. Les ions ont des énergies variant entre quelques dizaines de MeV à plusieurs centaines de MeV.

- Ceintures de radiation

Les ceintures de radiation sont constituées par des particules chargées, principalement des protons et des électrons, ainsi que par des éléments les plus légers de la table de classification périodique, des particules qui peuvent être piégées par le champ magnétique terrestre.

La figure 1.7 illustre les zones des particules piégées qui ont été découvertes en 1958 par J.A Van Allen (mission explorer I). Elles sont appelées ceintures de radiation ou ceintures de Van Allen et ont une forme toroïdale. Ces ceintures sont situées en dehors de l'atmosphère terrestre mais dans la zone de l'espace influencée par le champ magnétique de la terre. Leur énergie se situe entre une dizaine de KeV et quelques centaines de MeV.

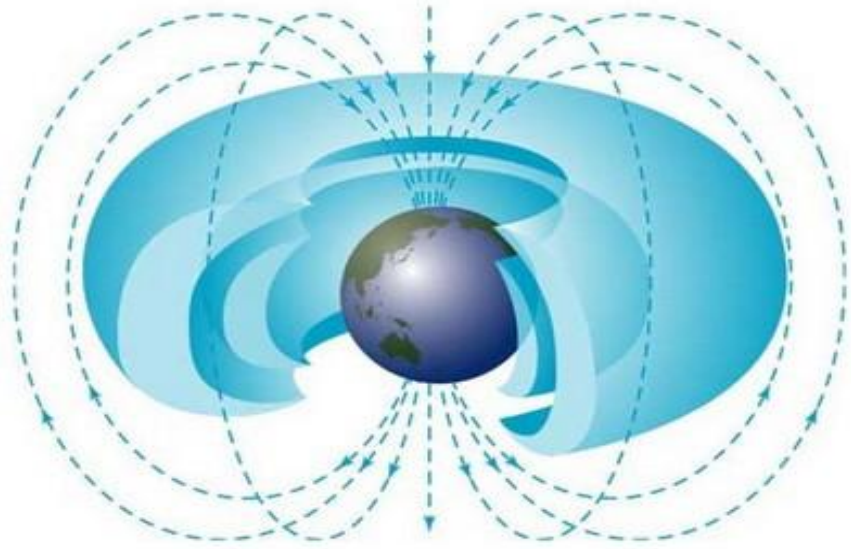


Figure 1.7 : Les ceintures de radiation de Van Allen.

Elles sont constituées de deux zones distinctes, appelées "ceinture intérieure" et "ceinture extérieure".

On peut distinguer principalement trois ceintures :

- Deux ceintures formées d'électrons et ils sont situés à 9000 Km et 30000 Km.

- Une ceinture formée de protons et se trouve à 12000 Km [Boudenot E3950].

Ces ceintures ne sont pas uniformes, présentant donc des anomalies locales, à cause de la déformation de la magnétosphère sous l'effet du vent solaire, du décalage et de l'inclinaison de l'axe magnétique et de l'axe de rotation terrestre. On peut noter que l'une des distorsions la plus importante se situe dans l'hémisphère Sud, au dessus de l'océan Atlantique. Cette région est particulièrement riche en protons, elle est appelée Anomalie Sud-Atlantique (SAA), figure 1.8.

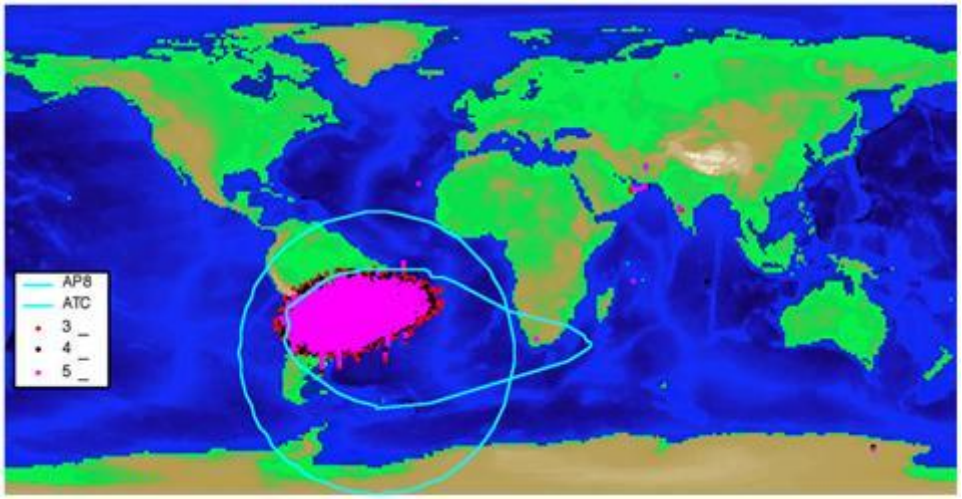


Figure 1.8 : Anomalie Sud-Atlantique

Le tableau 1.1 résume l'origine et la nature des particules rencontrées dans l'espace ainsi que leurs énergies et flux dans la magnétosphère [Delarochette 1995], [Boudenot 1995]. Il permet d'avoir une vue d'ensemble des rayonnements ionisants qui sont susceptibles de rencontrer un composant lors d'une mission spatiale.

Tableau 1.1: Environnement radiatif spatial.

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

En guise de conclusion, la figure 1.9 résume les trois types de radiation (figure 1.9) :

- Les particules à fort débit mais ayant faible énergie, 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.

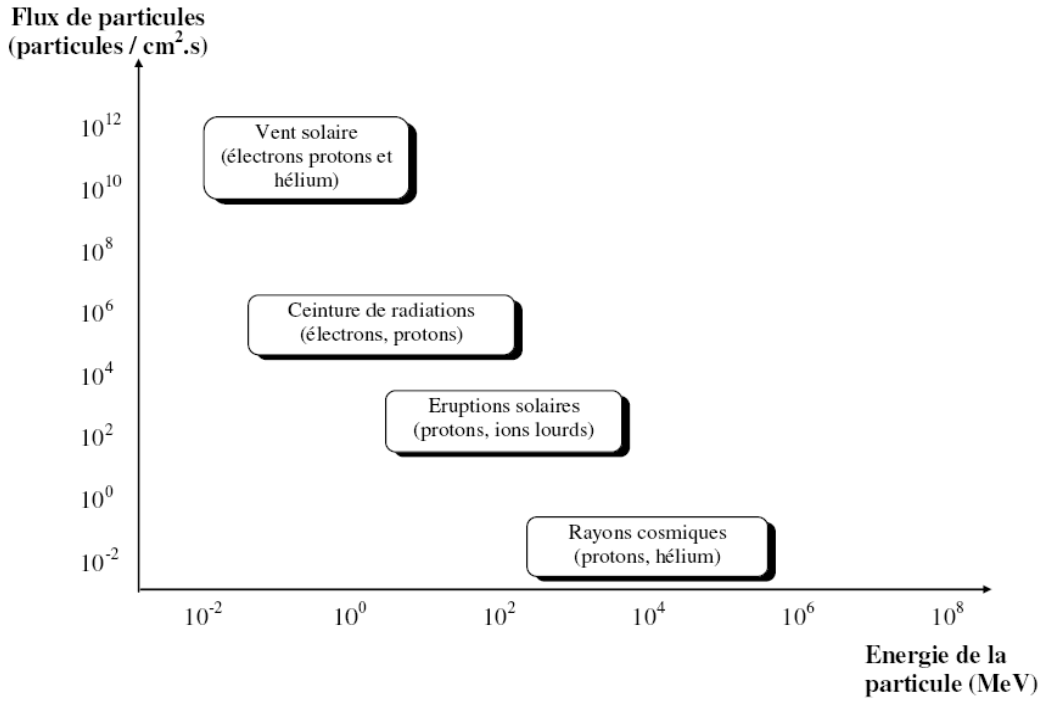


Figure 1.9 : Description des différents composants de l'environnement radiatif spatial. Variations du flux en fonction de l'énergie des particules

1.2.2 Environnement radiatif atmosphérique

L'environnement radiatif atmosphérique est principalement dû à l'interaction des rayonnements cosmiques, composés essentiellement d'hélium, de protons, d'électrons et des ions lourds, avec les atomes -Azote et Oxygène- présents dans l'atmosphère.

Cette interaction donne naissance à des nouvelles particules qui peuvent être des protons, photons, électrons, neutrons, ions lourds, muons et puons. Cette création se fait dans deux manières: soit par perte d'énergie par ionisation, soit par des réactions nucléaires formant une «douche » de particules secondaires comme montre la figure 1.10.

Avec l'avancement de la technologie VLSI et l'utilisation des transistors dont les gravures sont très fines, les circuits intégrés deviennent sensibles aux particules ayant des énergies inférieures à 100 MeV [Taber 1992] [Taber 1993]. Les pions de très faibles énergies ont une part négligeable dans la perturbation du fonctionnement des circuits intégrés. Ce n'est pas du tout le cas pour les ions lourds

et les neutrons. En fait, les neutrons sont les particules prédominantes aux altitudes avioniques. Ces particules sont non ionisantes et ont des énergies supérieures à 100 MeV. Elles peuvent entrer en collision avec les atomes constitutifs des différentes couches des composantes électroniques générant des ions qui peuvent induire des défaillances. Le flux moyen des neutrons au sol est estimé à 36 particules/cm^2/heure, et dépend de plusieurs facteurs tels: l'altitude, la latitude, le cycle solaire et le blindage environnant. Parmi ces neutrons, les flux qui contribuent à des perturbations dans des signaux et des données des circuits intégrés, perturbations appelées SEE (Single Event Effects) ou soft-erreurs, sont estimés entre 13 et 20 particules/cm^2/heure [JESD89 2001]. La figure 1.11 décrit les particules existant dans l'atmosphère en fonction de l'altitude [Brien 1971] [Brien 1978].

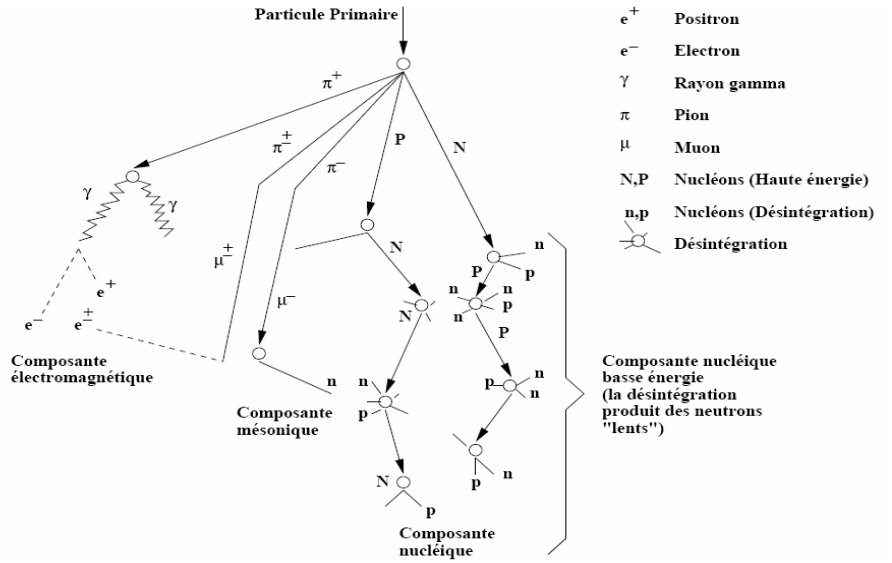


Figure 1.10 : Représentation de la génération de particules secondaires dans l'atmosphère

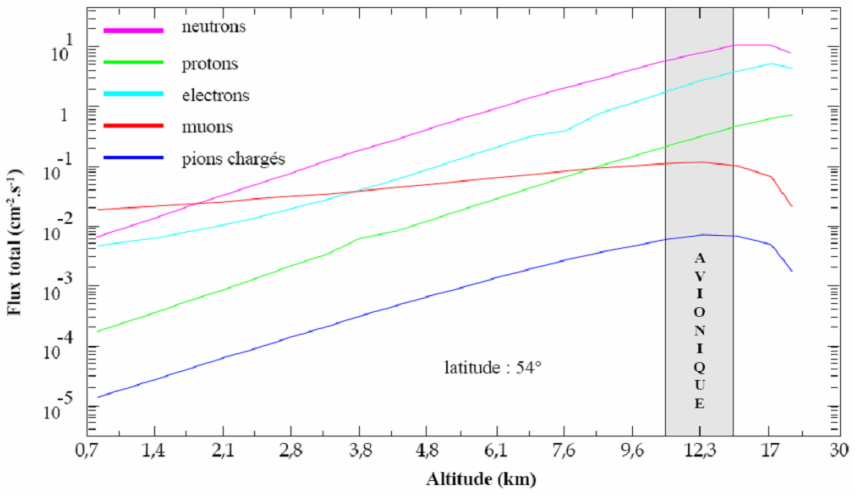


Figure 1.11 : Flux total de particules présentes dans l'atmosphère en fonction de l'altitude

1.3 Effet des radiations sur les circuits intégrés

Les particules énergétiques peuvent provoquer des effets transitoires, permanents ou destructifs dans les circuits intégrés qu'elles traversent. Cette section a pour but de présenter les mécanismes d'interaction entrant en jeu durant l'interaction particule-matière et les deux principaux types d'effets considérés, l'effet de dose et les effets singuliers.

1.3.1 Mécanisme d'interaction

La perte d'énergie d'une particule incidente lors de son passage dans la matière est en proportionnalité directe avec les dommages permanents ou transitoires. Cette quantité perdue, appelée « pouvoir d'arrêt total », est exprimée par dE/dX (MeV/cm) et étant composée de deux phénomènes principaux: la perte d'énergie électronique et la perte d'énergie nucléaire.

L'ionisation due aux interactions entre la particule incidente et les électrons des atomes du milieu traversé est la cause principale du premier phénomène. Les pertes électroniques vont aboutir à la création de paires électrons-trous le long du trajet de la particule. Ce pouvoir d'arrêt électronique est appelé LET (Linear Energy Transfer).

Le deuxième phénomène correspond aux collisions entre la particule et les noyaux provoquant la perte nucléaire d'énergie. Cette interaction peut conduire à l'éjection du noyau d'un réseau cristallin. Des défauts sont alors créés. La perte d'énergie associé à ce type de dommage est appelée perte d'énergie non ionisante, NIEL (Non Ionising Energy Loss).

On peut donc conclure que le pouvoir d'arrêt total peut s'écrire comme la somme de ces deux pertes d'énergies :

$$\frac{dE}{dX} = \left(\frac{dE}{dX}\right)_{elec} + \left(\frac{dE}{dX}\right)_{nuc} \qquad \text{Eq 1.1}$$

1.3.2 Effet de Dose

La dose est l'énergie par unité de masse déposée dans un matériau et s'exprime en Gray. Un Gray est équivalent à l'absorption d'un joule par kilogramme de matière.

En général, une grande partie des effets des radiations que l'ont 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. Quand une particule traverse un matériau, l'énergie perdue par celle-ci peut être cédée aux électrons du matériau traversé, c'est la dose ionisante. Si les atomes sont déplacés, on dit que la dose est non ionisante.

- Dose ionisante

L'énergie perdue par les particules incidentes peut être cédée aux électrons du matériau traversé. Selon la nature du matériau, les électrons peuvent atteindre la bande de conduction et ainsi libérer des trous dans la bande de valence [Oldham 2003]. Dans le cas du silicium légèrement dopé on admet que la génération des paires électrons-trous est équivalente à la densité des porteurs de charges à l'équilibre. L'effet est donc transitoire puis il s'éclipsera. Dans le cas d'un isolant l'effet est plus critique. Deux mécanismes qui affectent les composants électriques sont:

- Piégeage des charges dans les isolants (figure 1.12).
- Migration des charges vers les interfaces isolant/semi-conducteur et création d'un canal permanent.

Ceci a plusieurs conséquences: dérive de la tension de seuil, augmentation du courant de fuite, diminution du gain, variation du courant inverse des diodes et variation de la tension de claquage.

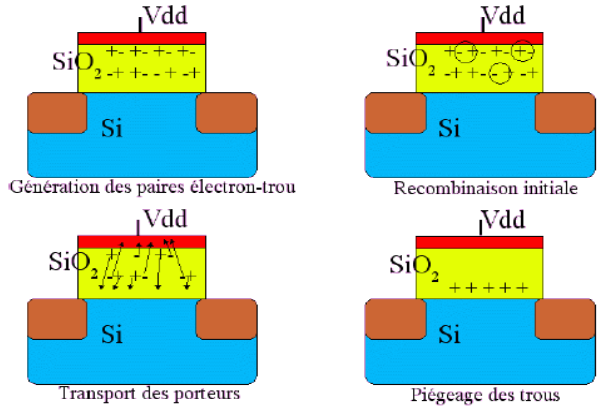


Figure 1.12 : Piégeage de trous par effets de dose.

- Dose non ionisante

L'introduction de défauts dans le réseau cristallin engendre de nouveaux pièges modifiant ainsi les caractéristiques de fonctionnement du circuit. Parmi les effets de dose non ionisante on peut citer les cinq suivants:

- Augmentation du courant de fuite.
- Modification du dopage du semi conducteur.
- Piégeage de porteurs.
- Effet tunnel.
- Réduction de la durée de vie des porteurs minoritaires.

Il faut bien noter que ces effets affectent les composants optoélectroniques ainsi que les transistors bipolaires qui sont particulièrement sensibles à ce type de dommage.

1.3.3 Effets singuliers

Les SEE, ont pour origine le même phénomène que celui de la dose ionisante, c.à.d. l'ionisation localisée le long de la trajectoire de la particule incidente. Ils résultent principalement de la déposition et de la collection de charges dans un nœud sensible du circuit. Les ions énergétiques qui traversent ce volume produisent directement une colonne d'électrons-trous qui peuvent provoquer un SEE. Un autre mécanisme qui peut ioniser la matière et conduire à un SEE, est dû aux sous-produits des collisions des protons et des neutrons en transférant toute ou une partie de leur énergie.

Deux phénomènes entrent dans le déclenchement d'un SEE: la génération des charges et la collection des charges [Duzellier 2004].

- Génération des charges

Comme dans la dose ionisante, on peut utiliser le LET pour exprimer la quantité des charges déposée par le passage d'un ion. La courbe de Bragg donne la représentation du LET en fonction de la distance parcourue par la particule (figure 1.13), on peut constater que le LET reste relativement constant durant la majeure partie du parcours de l'ion. Vers la fin du trajet, lorsque la particule perd presque toute son énergie, le LET augmente d'une façon brutale pour s'annuler rapidement lorsque la particule est au repos. Cette phase est appelée *PIC de Bragg*.

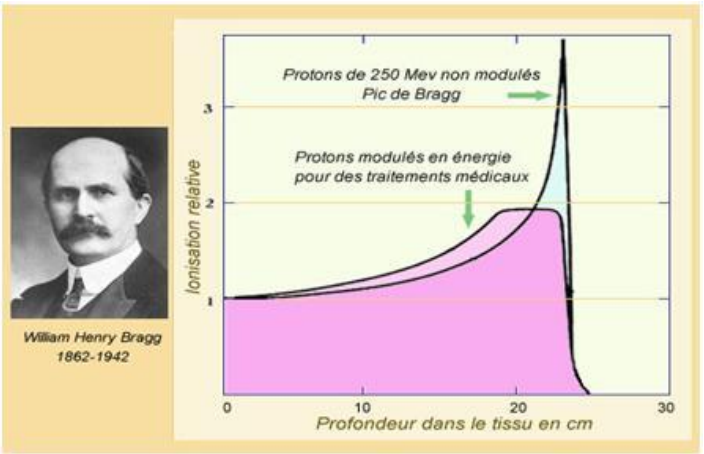


Figure 1.13 : Courbe de Bragg

- Collection des charges

Lors du passage d'une particule le long de sa trajectoire, une ionisation se produit résultant en des effets singuliers. Deux phénomènes à des instants différents participent à la collection des charges :

- Conduction (drift) : une durée inférieure à la nanoseconde est suffisante pour que les électrons se déplacent vers le drain du transistor à cause du champ électrique présent dans la zone de déplétion.
- Diffusion : les électrons générés dans le substrat peuvent diffuser vers le drain et être collectés. Ce phénomène peut durer plusieurs nanosecondes.

Un troisième phénomène appelée « *funneling* » peut entrer en jeu lorsque l'ion quitte la zone de déplétion. Il distord le champ électrique et l'étire en dehors de cette zone. La figure 1.14 illustre ces phénomènes.

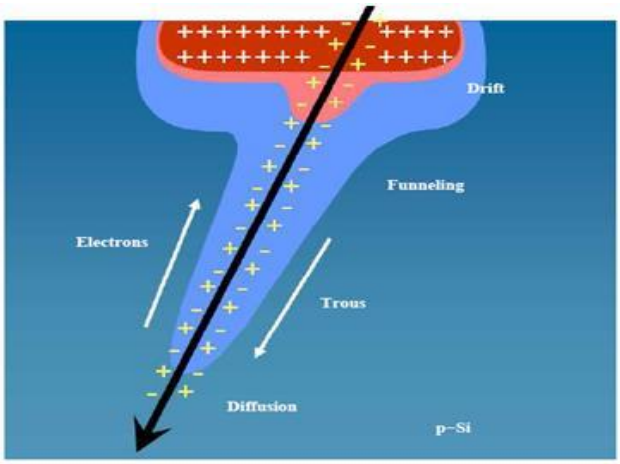


Figure 1.14 : Mécanismes de collection de charges : drift, funneling et diffusion.

Différents types d'événements peuvent se produire comme conséquence de l'impact d'une particule ionisante:

- SET (Single Event Transient)

La particule incidente induit par ionisation une quantité de charges. Cette charge est collectée par le champ électrique ce qui résulte en la création d'un photocourant. Les conséquences de ce type d'effet sont principalement la génération de pulses de courants indésirables qui peuvent perturber le fonctionnement des systèmes numériques et analogiques.

- SEU (Single Event Upset)

Le photocourant crée un changement d'état dans une structure de type « bascule ». Ce phénomène se produit lorsque la charge induite par la particule dépasse la quantité nécessaire au changement d'état. Cet évènement est non destructif, peut générer des erreurs en perturbant le contenu des cellules mémoires (mémoires SRAMs, bascules, registres et les circuits numériques séquentiels).

- SEL (Single Event Latchup)

Ce phénomène se caractérise par le déclenchement du thyristor parasite (pnpn parasite) inhérent à la structure CMOS. Un verrouillage fonctionnel, parfois destructif peut avoir lieu.

- SEFI (Single Event Functional Interrupt)

Cet effet se traduit par le blocage de la fonctionnalité du composant en le faisant passer dans des états interdits. Ce phénomène concerne notamment les machines à états des composants numériques.

- SEGR (Single Event Gate Rupture)

Le dépôt de charge conduit à la destruction d'une structure MOS par le claquage de l'oxyde de grille.

- MCU (Multiple Cell Upset)

Une seule particule provoque le basculement de plusieurs points mémoires physiquement voisins.

- MBU (Multi Bit Upset)

Ce phénomène prend place quand la particule crée plusieurs erreurs dans un même mot mémoire. Sa probabilité est plus faible que celle du SEU mais elle croît avec la miniaturisation de l'électronique. Il en résulte une complexification de la problématique de détection et de correction d'erreur dans les systèmes numériques.

- SHE (Single Hard Errors)

La particule provoque un basculement irréversible d'un point mémoire.

- SEB (Single Event Burnout)

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 ces voisins par effet d'avalanche et rendre le circuit inutilisable.

Cette thèse se concentre sur les effets de type SEU, SET, MCU et MBU ayant lieu dans des circuits intégrés tels que les microprocesseurs, les processeurs et les composantes de type SRAMs.

1.4 Sensibilité des circuits avancés aux neutrons atmosphériques

Dans le but d'obtenir un retour objectif sur la criticité de la conjoncture « sensibilité des circuits intégrés avancés face aux effets des radiations présentes dans l'atmosphère terrestre », l'un des

objectifs de cette thèse a été de participer à la réalisation des expérimentations en environnement réel. Une plateforme de test expérimentale dédiée à la détection d'événements singuliers a été activée lors de vols commerciaux de longue durée, dans des ballons stratosphériques de courte durée ainsi que dans des montagnes et des villes situées à hautes altitudes. Des résultats obtenus par prédiction effectuée à l'aide de l'outil MUSCA SEP^3/ADDICT (Multi-SCAles Single Event Phenomena Predictive Platform) développé à l'ONERA [Hubert 2009] ont été confrontés à ceux issus de cette plateforme.

Le FIT (Failure In Time) est l'unité utilisée pour exprimer le taux d'erreurs, SER (Soft-Error Rate), des circuits intégrés. Un FIT est égal à une erreur par milliard heures (10^9 h ce qui correspond à environ 114155 années). Par exemple, le SER d'une mémoire SRAM est estimé entre 200 et 2000 FIT [Web cisco], le SER d'une mémoire DRAM opérant à une vitesse maximale est estimé entre quelques centaines et quelques milliers de FIT, celui d'une SRAM en technologie très ancienne (<0.25 micron) est estimé entre 10000 et 100000 FIT [Leung 2000]. La détermination du FIT de circuits tels que les FPGAs et les SRAMs fonctionnant à différentes altitudes a été l'objectif de plusieurs expériences présentées dans l'état de l'art faites depuis les années 90s [Olsen 1993] [Normand 1996] [Lesea 2005]. Ces expériences ont été les premières à être réalisées dans ce domaine et ont apportées des données objectives sur la sensibilité des semi-conducteurs aux effets des particules présentes dans l'atmosphère terrestre. Le but de certaines de ces expériences était la validation de l'efficacité des solutions de durcissement au niveau de conception, pour des composants destinés à être utilisés dans des applications critiques (systèmes avioniques…).

Le faible flux de particules présentes dans l'atmosphère terrestre impose dans le cas de telles expériences l'implémentation d'un grand nombre de circuits augmentant la surface sensible et donc la probabilité d'observer des soft-erreurs. Les procédés de fabrication disponibles de nos jours ainsi que les technologies à faible consommation permettent le développement des plateformes expérimentales possédant une surface sensible significative avec un nombre réduit de composants.

1.4.1 La plateforme expérimentale

La plateforme expérimentale utilisée au cours de cette thèse a été développé à TIMA dans le cadre d'une thèse précédente [Peronnard 2009]. Elle embarque 1 Gbit de mémoire construite à partir de 64 chips SRAM. Ces chips proviennent de deux générations successives d'un circuit mémoire 130nm et 90nm (respectivement CY62167DV30LL et CY62167EV30LL) [Web cypress]. Des plateformes toutes en 130nm, toutes en 90nm, et mixtes ont été fabriquées et utilisées durant ces expériences pour obtenir des données mettant en évidence l'impact, sur la sensibilité aux soft-erreurs, de la technologie de fabrication. Pour permettre son utilisation dans divers contexte (avion, ballon,

montagnes etc…), cette plateforme peut être alimentée soit par une batterie rechargeable soit par un adaptateur de tension de 5V.

La carte est basée sur un FPGA Spartan3 de Xilinx qui assure la communication avec le PC. Ce FPGA est responsable de fournir les adresses aux SRAMs ainsi que les 16-bits de données. Le choix du chip est contrôlé par un CPLD CoolRunnerII après le décodage du numéro de chip provenant du FPGA. La figure 1.15 montre l'architecture de la carte alors que sa photo est montrée dans la figure 1.16.

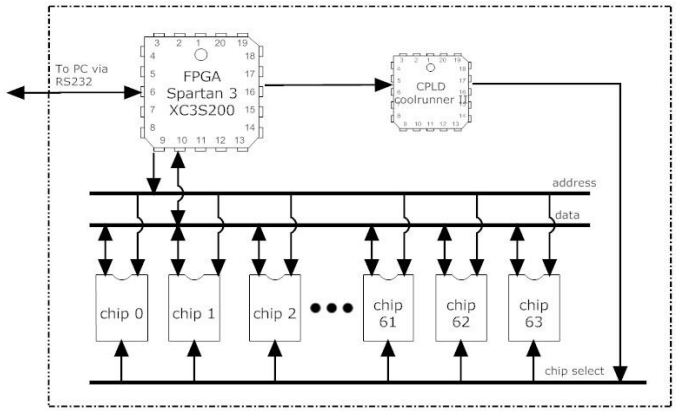


Figure 1.15 : Architecture de la plateforme de test

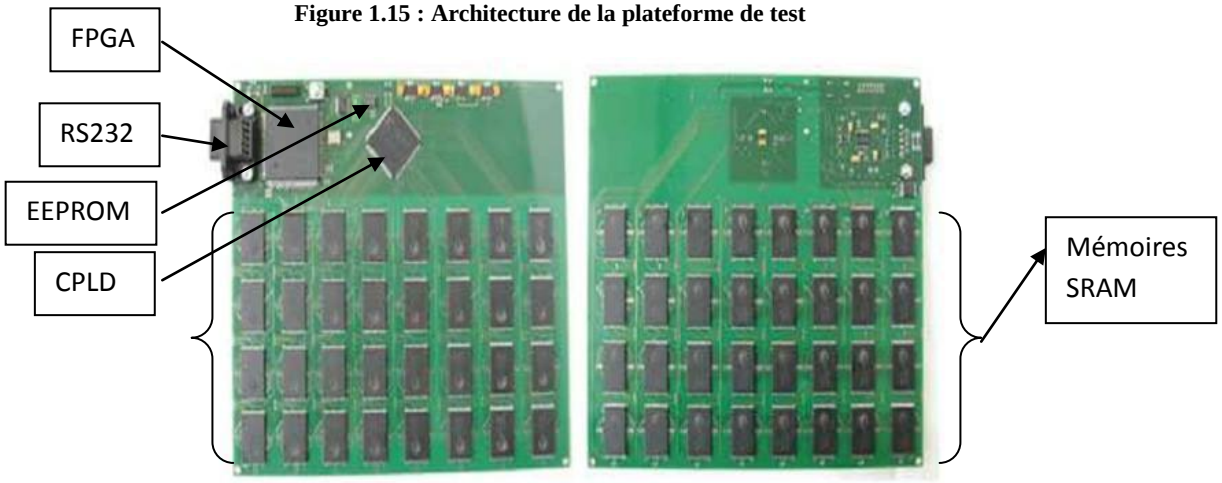


Figure 1.16 : Carte SRAMs développée pour les tests en environnement réel

Cette carte est programmée pour exécuter les étapes suivantes :

- Écriture du motif de référence (par exemple 0x5555).
- Attente, une minute.
- Lecture et vérification du contenu.
- Rapport d'erreurs.

Un lien série RS232 assure la communication entre la carte et un PC directement connecté ou à distance. En cas de détection d'erreur, le numéro du circuit, l'adresse et la donnée erronée sont reportés. Le CPLD est utilisé pour l'adressage du plan mémoire. Il s'agit d'un décodeur 1 à 64 bits et permet ainsi de lire chacune des mémoires présentes sur la carte.

Plusieurs modifications ont été faites sur cette carte durant cette thèse. Une horloge à temps réel permettant de connaître l'instant d'occurrence des évènements (avec la marge d'une minute environ) est implémentée. Une mémoire non volatile EEPROM (existante sur la carte mais non utilisée) a été activée pour stocker les erreurs durant les longs vols, dans les ballons et dans les zones situées à hautes altitudes pour ne pas perdre les informations sur les erreurs détectées en cas de coupure d'alimentation. De plus, le logiciel utilisé pour le pilotage de la carte a été mis à jour afin que toutes les informations sur les erreurs, en cas de détection des soft-erreurs, soient automatiquement transmises par e-mail. Le nouveau logiciel est capable d'auto-injecter des fautes dans tous les circuits mémoires afin de valider de façon périodique (chaque heure) le bon fonctionnement de la plateforme.

Tableau 1.2: Erreurs détectées durant des vols commerciaux de longue durée

Départ	Arrivée	Minute	Chip	Adresse	Donnée lue en hexa (0x5555)
Madrid	**Buenos Aires**	N/A	0x3B	0x657F6	0x5557
Madrid	**Buenos Aires**	N/A	0x3B	0x657FA	0xD557
Madrid	**Buenos Aires**	N/A	0x3B	0x657BE	0xF557
Paris	**Lima**	199	0x26	0xDE78B	0x55D5
Paris	**Lima**	199	0x26	0xDE68B	0x55D5
Paris	**Lima**	199	0x26	0xC178A	0x5554
Paris	**Sao Paulo**	1041	0x17	0xEE6CC	0xD555

Des erreurs provoquées par les radiations présentes dans l'atmosphère terrestre ont été détectées lors de l'activation de ces plateformes expérimentales durant plusieurs vols commerciaux de longue durée entre l'Europe et l'Amérique du Sud. Dans le tableau 1.2 sont montrés quelques résultats significatifs obtenus avec la version précédente de la plateforme, dans un vol (Madrid-Buenos Aires), ainsi que ceux obtenus avec la nouvelle version lors de deux vols (Paris-Lima et Paris-Sao Paolo). Dans le vol (Madrid-Buenos Aires) les erreurs détectées sont des erreurs de multiplicité variable. La première erreur est clairement un SEU, tandis que la seconde et la troisième sont des erreurs de multiplicité 2 et 3 ayant lieu dans un seul mot (MBU). Elles sont dues à une seule particule puisque les adresses sont en voisinage physique, donc il s'agit d'un MCU. Le lien entre l'adressage physique et logique a été confirmé par *reverse-engineering* réalisé en laser. Les SEUs détectés pendant le vol

Paris-Lima sont juste des MCUs puisque une seule particule a provoqué le basculement de deux cellules mémoires physiquement adjacent. Pour ce qui est du troisième SEU détecté dans le même cycle de lecture des mémoires (environ 3 heures et demi après le départ) et ayant lieu dans le même chip, il n'est pas possible de distinguer s'il provient de la même particule que les deux précédentes, ou bien d'une nouvelle particule. L'erreur détectée pendant le vol (Paris-Sao Paulo) est clairement un SEU.

1.4.2 MUSCA SEP^3/ADDICT : un outil de prédiction

La modélisation multi-échelles et la physique liée à l'occurrence des SEE dans l'environnement naturel ont fait l'objet des recherches et d'outils développés dans plusieurs institutions, telles que l'ONERA [Hubert 2009] [Hubert 2010], l'Université Vanderbilt [Weller 2009] [Sierawki 2009] et IBM [Tang 2004]. Comme le montre la figure 1.17, MUSCA SEP^3 effectue une simulation Monte-Carlo des événements dus aux effets des radiations. Les sorties (charges générées) sont utilisés par ADDICT pour simuler la réponse électrique du dispositif, tel que décrit dans [Artola 2011]. Ces simulations sont basées sur une modélisation du courant transitoire généré en tenant compte des mécanismes de transport dynamique et de la collection des charges. La méthodologie utilisée permet la modélisation séquentielle pour tous ces différents mécanismes physiques.

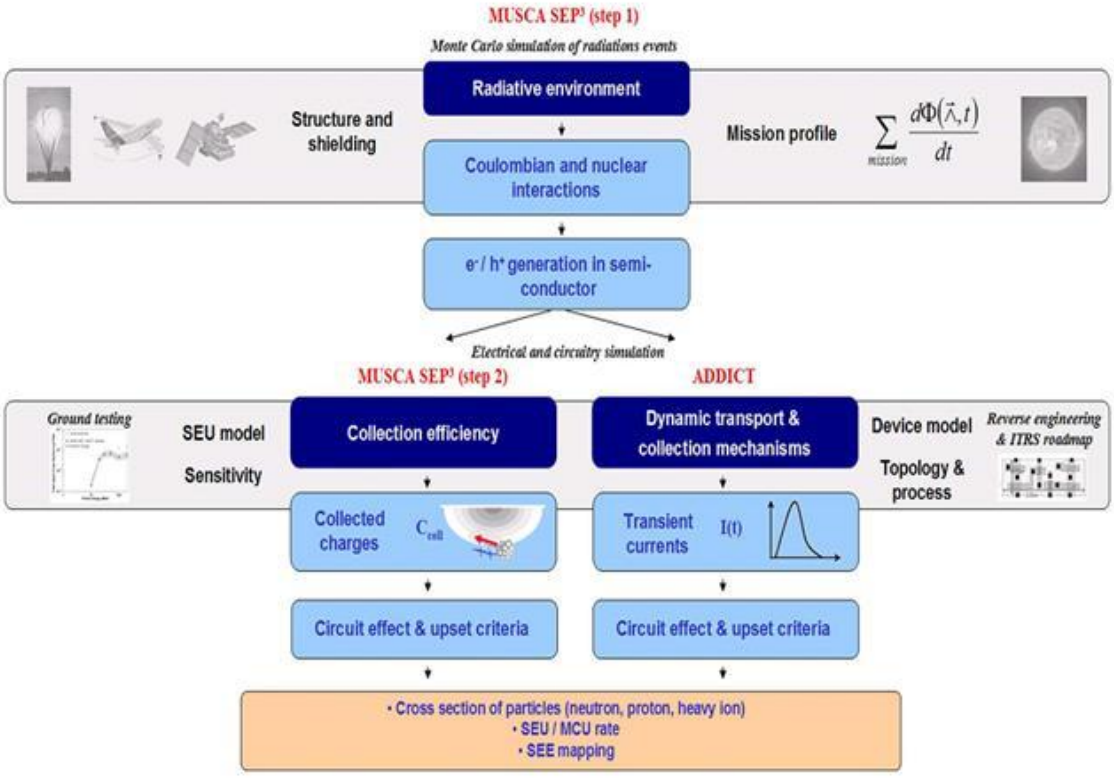


Figure 1.17: Méthodologie de la plateforme de prédiction MUSCA SEP^3/ADDICT

1.4.3 Résultats expérimentaux

1.4.3.1 Données obtenues en vols long courrier

Dans ce qui suit sont décrits les premiers résultats obtenus dans les recherches faites à TIMA dans le cadre d'une thèse précédente [Peronnard 2009]. Lors de l'activation de la première version de la plateforme constituée des mémoires SRAM à 130 nm dans un vol Los Angeles-Paris le 23/4/2009. Les erreurs détectées sont présentées en détails dans le tableau 1.3.

Tableau 1.3: Erreurs détectées durant un vol Los Angeles – Paris

Nb	Chip	Adresse	Donnée lue en hexa (0x5555)	
1	10 (0x0A)	0x37DF1	0x5755	(0101011101010101)
2	10 (0x0A)	0x3BDF1	0x5755	(0101011101010101)
3	10 (0x0A)	0x3BDF6	0x5755	(0101011101010101)
4	10 (0x0A)	0x37DDF	0xF555	(1111010101010101)
5	32 (0x20)	0x2535A	0x7555	(0111010101010101)
6	32 (0x20)	0x3535B	0x4555	(0100010101010101)
7	32 (0x20)	0xB525B	0x4555	(0100010101010101)
8	41 (0x29)	0x81E5E	0x5155	(0101000101010101)
9	42 (0x2A)	0xD7B31	0x1555	(0001010101010101)
10	52 (0x34)	0xF0983	0x5557	(0101000101010111)
11	53 (0x35)	0x6A33F	0x5D55	(0101110101010101)
12	53 (0x35)	0x7A33E	0x5D55	(0101110101010101)
13	55 (0x37)	0xAE5CB	0xD555	(1101010101010101)
14	56 (0x38)	0x919B9	0x7555	(0111010101010101)

Ce vol fut choisi puisqu'il a permis d'observer un nombre important d'erreurs de multiplicité diverses. En effet, 15 soft-erreurs ont été détectées dont 5 SEU (voir lignes 8, 9, 10, 13 et 14), 3 MCU ((1, 2, 3, 4), (5, 6, 7) et (11, 12)) et 1 double MBU (4)). Le voisinage des adresses des MCU détectés dans les lignes 1, 2 et 3 est clairement justifié par une proximité physique 0x37DF1, 0x3BDF1 et 0x3BDF6. L'adresse de l'erreur, double MBU, détectée dans la ligne 4, dans le même chip, ne peut pas être corrélée avec les adresses des erreurs précédentes. Pour conclure sur la nature de l'erreur, l'impact de l'absence, dans cette version de la plateforme, d'informations concernant la période d'occurrence est clairement mise en évidence. On peut conclure que cette erreur (ligne 4) peut provenir de la même particule qui a provoqué les erreurs précédentes ou bien d'une autre particule impactant le même chip.

Dans le deuxième MCU détecté (voir ligne 5, 6 et 7), un voisinage physique entre les adresses peut être distingué malgré le manque des informations sur le décodage des adresses dans un chip.

Dans le troisième MCU (lignes 11 et 12), les adresses sont très probablement voisines (0x**6**A33**F** et 0x**7**A33**E**).

Ces résultats ont été utilisés pour la validation de l'outil de prédiction MUSCA SEP^3/ADDICT. Le nombre des soft-erreurs intégrés pour la durée du vol, (triangles rouges) est tracé sur la figure 1.18. Le différentiel du flux de particules impliqué pendant le vol a été simulé avec l'outil QARM (Qinetiq Atmospheric Radiation Model) [Web geoshaft]. La confrontation des résultats observés et ceux prédits montre que le nombre d'erreurs détectées, 15 soft-erreurs (triangle rouge) est correctement couvert par la prédiction (zone bleue). En plus, des estimations précédentes sont données par la barre jaune foncé.

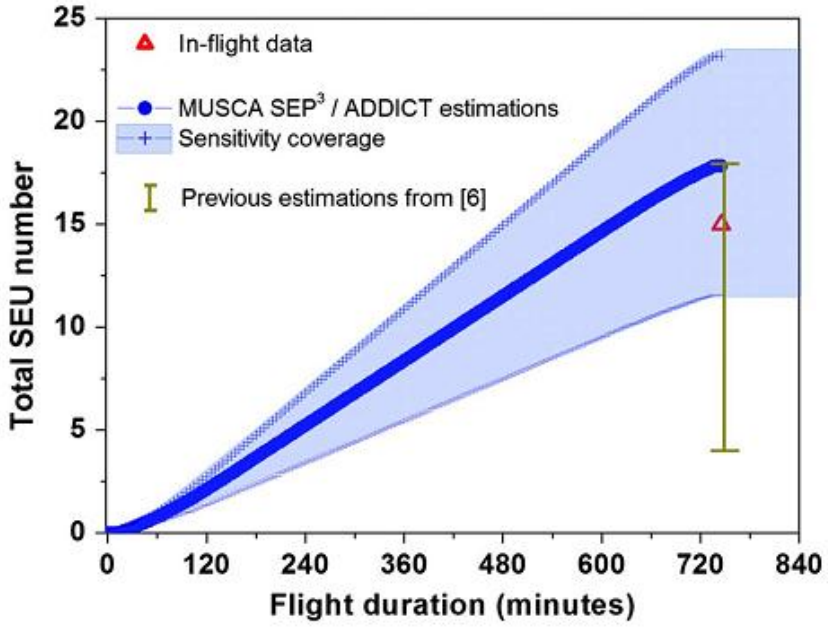


Figure 1.18: Le nombre des SEUs détectés pendant un vol Los Angeles - PARIS Vs. SEUs prédits par MUSCA SEP^3/ADDICT

1.4.3.2 Données issues des vols en ballons stratosphériques

La plateforme expérimentale bâtie sur des mémoires Cypress en 90nm a été incluse en une *piggy-back* lors d'un vol ballon lancé par le CNES en 2010 à Kiruna, au Nord de la Suède (67°51'00"N, 20°13'00"E). Cinq vols ballons ont été effectués pour une durée de vol totale d'environ 45 heures. Des événements SEE ont été détectés au cours d'un seul vol. Ce vol à duré environ 6 heures à une altitude moyenne de 25~30 Km et avec une altitude maximale d'environ 43 Km.

La figure 1.19 représente le tracé du vol ballon lancé en 2010 en Suède. Le ballon s'est rendu à environ 100 Km au Nord Ouest. Les données de localisation ont été fournies par un GPS à bord des expériences.

Figure 1.19: Carte du cours du vol ballon lancé près de Kiruna (Suède) en 2010

L'environnement radiatif, modélisé par QARM et caractérisé par un flux de particules mixte de neutrons et de protons dont leurs spectres d'énergie pour deux altitudes représentatives du profil du vol ballon lancé en 2010 à Kiruna, est donné par la figure 1.20.

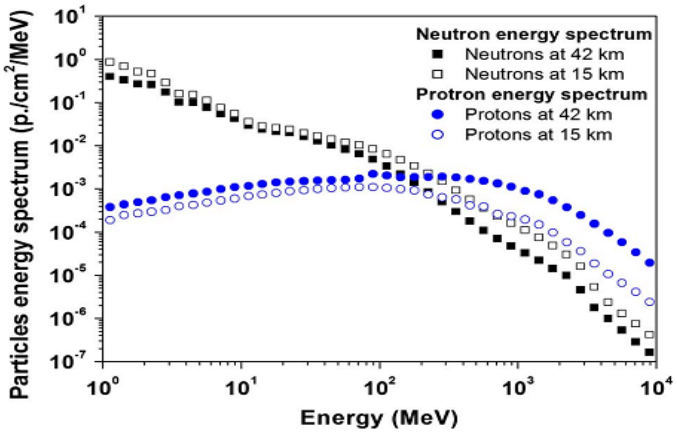


Figure 1.20: Spectres d'énergies des protons et des neutrons modélisés par QARM pour 42 et 15 Km

Les données recueillies par la plateforme expérimentale activée pendant ce vol ballon, ont montré de nombreuses erreurs dans des cellules mémoires, une majorité d'entre eux ayant été provoquées par une unique particule affectant des cellules voisines (MCU). 13 soft-erreurs (5 SEU et 7 MCU) ont été observées durant ce vol.

Le calcul effectué par la plateforme de prédiction MUSCA SEP^3/ADDICT a été réalisé avec le modèle moyen pertinent (dispositif et sensibilité) pour donner une prédiction de types d'erreurs qui peuvent être détectées par la plateforme expérimentale. La contribution des erreurs simples et

multiples par rapport au totale des soft-erreurs mesurés par la plateforme expérimentale est résumée dans la figure 1.21.

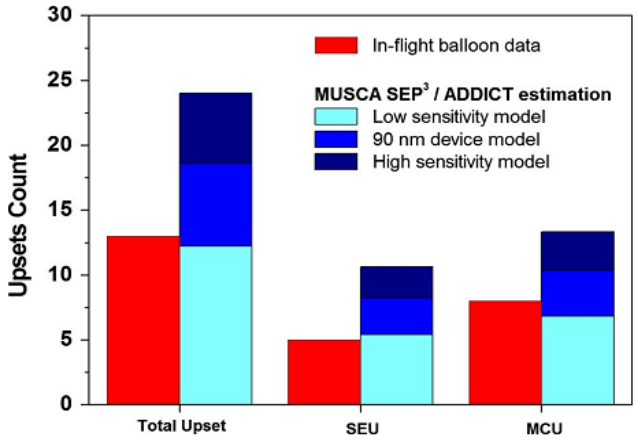


Figure 1.21: Comparaison du nombre des SEUs prédits et mesurés lors du vol ballon

Le nombre d'erreurs observées, 13 soft-erreurs (barres rouges), est correctement couvert par les calculs prédictifs. La prédiction pour des circuits en 90 nm (barres bleues) validée par l'outil de prédiction MUSCA SEP^3/ADDICT annonce 18 soft-erreurs comme moyenne.

La contribution des erreurs simples et multiples au nombre totale des soft-erreurs est bien estimée, comme la montre la figure 1.21. Les modèles de sensibilité basse et haute correspondent à une variation de ±25% de la charge critique (0,1 fC) du model du dispositif à 90 nm.

Dans la figure 1.22 le SER estimé pour la plateforme expérimentale pendant les 6 heures du vol ballon est représenté dans la figure 1.22. A plus de 30 km d'altitude, la contribution aux soft-erreurs des neutrons et des protons est similaire.

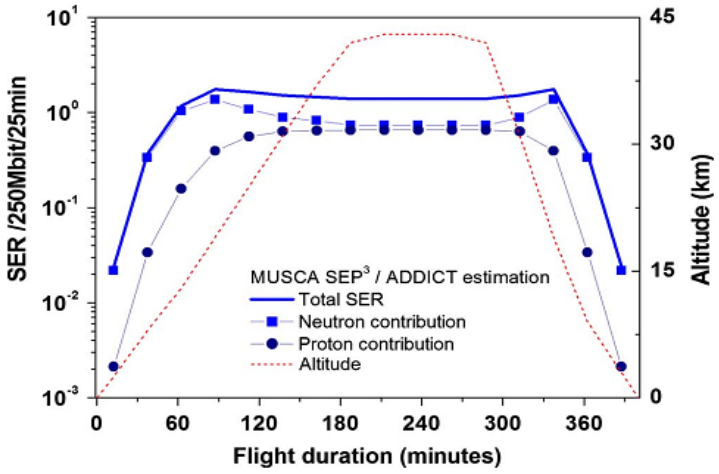


Figure 1.22: Le SER prédit à bord du ballon stratosphérique

1.4.3.3 Données des expériences à hautes altitudes

La plateforme expérimentale a été installée dans des montagnes et des villes à hautes altitudes pendant des longues durées pour obtenir des données expérimentales objectives sur la sensibilité aux soft-erreurs des mémoires SRAMs. Puno (-15° 50' 30.96"N, -70° 1'27.20E"), ville située au Sud du Peru ayant une altitude de 3800 m, figure 1.23, a été choisie comme première destination pour ces tests. La mémoire SRAM de 1Gbit de cette carte est bâtie avec des circuits Cypress issus de deux générations successives des chips (16 chips à 90 nm et 48 chips à 130 nm chacun des chips possède 2^{24} bits organisés en 2^{20} mots de 16 bits).

L'une des contributions importante de ces travaux a été la modification du logiciel de la carte pour permettre l'envoi automatique via internet dans les cas où les erreurs sont détectées. Durant les 5 mois quand duré ces tests, un événement par semaine de type SEU et MCU a été détecté. La multiplicité des MCUs détectés varient entre deux et six. Des telles erreurs ayant lieu dans des mots différents ne sont pas critiques si des techniques de tolérance aux fautes sont appliquées, telles que le code Hamming et la parité.

Le tableau 1.4 montre les SEUs détectés lors de cette mission terrestre entre mars et juillet 2012, mettant en évidence la criticité de ce genre de problème. Les erreurs qui sont potentiellement la conséquence des MCU sont indiquées avec la même couleur. En effet, elles ont lieu dans le même cycle de lecture, donc il est très hautement probable qu'ils sont la conséquence de l'impact d'une unique particule.

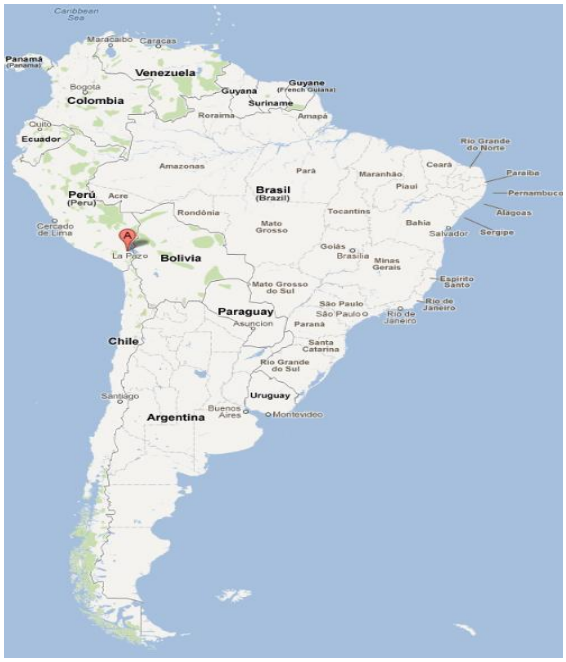


Figure 1.23: La localisation de Puno sur la carte

Tableau 1.4: SEUs détectés à PUNO, Peru

Nb	Date (heure local France)	Chip	Adresse (hexa)	Données lues
1	15/03/2012 12:23:32	45 (0x2D) 130 nm	0xF5166	0x555D (0101010101011101)
2	20/03/2012 08:52:15	60 (0x3C) 130 nm	0x434BA	0x1555 (0001010101010101)
3	23/03/2012 21:31:34	53 (0x35) 130 nm	0x32D2A	0x555D (0101010101011101)
4	26/03/2012 15:08:21	57 (0x39) 130 nm	0x9593A	0x5155 (0101000101010101)
5	26/03/2012 15:08:21	57 (0x39) 130 nm	0x8593B	0x5155 (0101000101010101)
6	26/03/2012 15:08:21	57 (0x39) 130 nm	0x5583A	0x5155 (0101000101010101)
7	01/04/2012 08:21:59	11 (0x0B) 90 nm	0x1BF6A	0x55D5 (0101010111010101)
8	01/04/2012 08:21:59	11 (0x0B) 90 nm	0x17F6A	0x55D5 (0101010111010101)
9	03/04/2012 17:46:12	44 (0x2C) 130 nm	0x20654	0x5554 (0101010101010100)
10	05/04/2012 13:13:06	34 (0x22) 130 nm	0x5FE91	0x7555 (0111010101010101)
11	13/04/2012 01:50:09	40 (0x28) 130 nm	0xD2253	0x4555 (0100010101010101)
12	13/04/2012 01:50:09	40 (0x28) 130 nm	0x52353	0x4555 (0100010101010101)
13	07/05/2012 10:33:43	24 (0x18) 130 nm	0xB41CB	0x55D5 (0101010111010101)
14	29/05/2012 16:50:23	09 (0x09) 90 nm	0x2DCB7	0x1555 (0001010101010101)
15	29/05/2012 16:50:23	09 (0x09) 90 nm	0x2DCB5	0x1555 (0001010101010101)
16	29/05/2012 16:50:23	09 (0x09) 90 nm	0x23CB7	0x1555 (0001010101010101)
17	29/05/2012 16:50:23	09 (0x09) 90 nm	0x23CB5	0x1555 (0001010101010101)
18	02/06/2012 19:33:12	35 (0x23) 130 nm	0xBDA8C	0x5455 (0101010001010101)
19	02/06/2012 19:33:12	35 (0x23) 130 nm	0xADA8D	0x5455 (0101010001010101)
20	02/06/2012 19:33:12	35 (0x23) 130 nm	0x3DB8C	0x5455 (0101010001010101)
21	02/06/2012 19:33:12	35 (0x23) 130 nm	0x2DB8D	0x5455 (0101010001010101)
22	03/06/2012 12:40:12	19 (0x13) 90 nm	0x5D817	0x5554 (0101010101010100)
23	03/06/2012 12:40:12	19 (0x13) 90 nm	0x5D817	0x5554 (0101010101010100)
24	04/06/2012 02:45:56	09 (0x09) 90 nm	0x64199	0x5515 (0101010100010101)
25	04/06/2012 10:49:56	36 (0x24) 130 nm	0x96108	0x4555 (0100010101010101)
26	13/06/2012 09:14:25	60 (0x3C) 130 nm	0x7DC38	0x5545 (0101010101000101)
27	29/06/2012 20:34:01	29 (0x1D) 130 nm	0xC4075	0x5155 (0101000101010101)
28	29/06/2012 20:34:01	29 (0x1D) 130 nm	0x54075	0x5755 (0101011101010101)
29	09/07/2012 11:27:34	55 (0x37) 130 nm	0x12904	0x5554 (0101010101010100)
30	16/07/2012 02:32:07	20 (0x14) 130 nm	0x3CFCB	0x5557 (0101010101010111)
31	16/07/2012 02:32:07	20 (0x14) 130 nm	0x2CECA	0x5551 (0101010101010001)
32	17/07/2012 16:47:41	36 (0x24) 130 nm	0xD25B1	0x5515 (0101010100010101)
33	17/07/2012 16:47:41	36 (0x24) 130 nm	0xC25B1	0x5515 (0101010100010101)
34	17/07/2012 16:47:41	36 (0x24) 130 nm	0xC24B0	0x55D5 (0101010111010101)
35	17/07/2012 16:47:41	36 (0x24) 130 nm	0x525B1	0x5515 (0101010100010101)
36	17/07/2012 16:47:41	36 (0x24) 130 nm	0x324B1	0x5515 (0101010100010101)
37	17/07/2012 16:47:41	36 (0x24) 130 nm	0x225B0	0x55D5 (0101010111010101)

Les SEUs (voir numéros 1, 2, 3, 9, 10, 13, 24, 25, 26 et 29) détectés ayant eu lieu dans les mémoires 130 nm et 90 nm proviennent des différentes particules puisque leurs instant d'occurrence est unique. Pour le cas des erreurs (14, 15, 16, 17) qui ont eu lieu dans le chip 9 (en technologie 90 nm) et qui ont été détectées dans le même cycle de lecture correspondent est un MCU puisque les

adresses sont physiquement voisines, et le bit perturbé a le même index dans chacun des mots erronés identifiés. Pour ce qui concerne l'erreur sextuple (voir lignes 32, 33, 34, 35, 36 et 37) qui a été détectée dans le chip 36 (en technologie 130 nm), l'analyse est plus compliquée. En effet, l'instant d'occurrence est le même pour les six erreurs, mais les voisinages physiques n'étant pas connu, il n'est pas possible de conclure s'il s'agit d'une particule provoquant plusieurs erreurs ou des plusieurs particules. Le flux pas trop important des neutrons mesuré par un dosimètre à cette altitude dans le cadre du projet HARMLESS [1] à l'emplacement de cette expérience, environ 1 neutron/minute, rend faible la probabilité que ces erreurs soient dues à plusieurs particules [Federico 2012].

La distribution des erreurs sur les différentes technologies des chips ainsi que la multiplicité des MCUs sont montrés dans la figure 1.24.

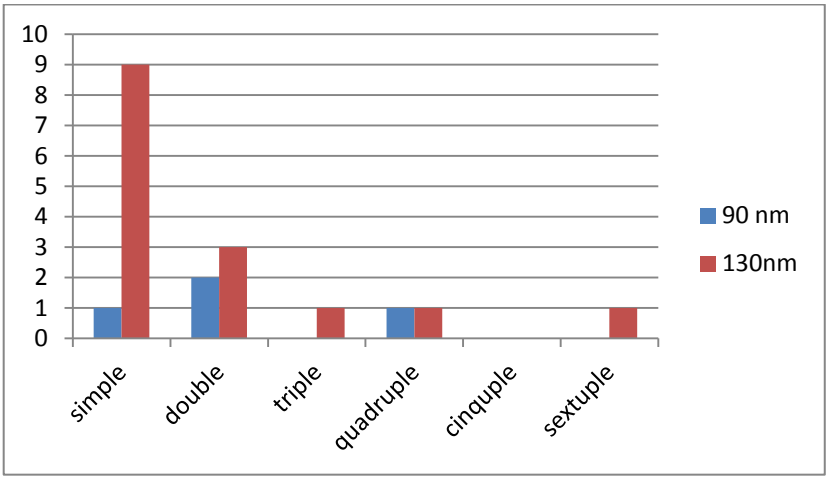


Figure 1.24: La répartition des erreurs sur les technologies des SRAMs

Ces résultats montrent que la technologie SRAM à 130nm serait plus sensible aux effets singuliers que celle à 90 nm. En effet 9 SEUs ont été détectés, 3 erreurs doubles, une triple, une quadruple et une sextuple. Tandis que dans la 90 nm 1 SEU a été détecté, 2 doubles et une quadruple. Cette tendance est justifiée par le nombre des chips implémentés sur la plateforme pour chacune des technologies (¼ des chips à 90 nm et ¾ des chips à 130 nm).

Le refuge des Cosmiques situé près de l'Aiguille du Midi dans la région de Chamonix (45°55'00"N, 06°51'00"E), en France, à l'altitude de 3613 m (figure 1.25), a été une deuxième cible pour ces recherches. La carte constituée des chips à 130 nm a été installée et activée en juin 2012 pour une courte période. La coupure du courant électrique et du réseau internet ont été des obstacles dans ces expériences. Les résultats reçus dans la période 24-28 juin sont montrés dans le tableau 1.5. Il est à

[1] Projet STIC-AMSUD, HARMLESS (High Altitude Remotely Monitored Laboratory for the Evaluation of the Sensitivity to SEU

noter qu'il a été utilisé pour cette expérience un dispositif [2] permettant la communication par ondes radio entre la carte de détection et l'ordinateur. De ce fait la plateforme expérimentale peut être installée à l'extérieur du bâtiment évitant l'atténuation du flux de neutrons lors de leur traversé du toit et des murs du bâtiment.

Figure 1.25: L'Aiguille du Midi prêt de Chamonix, France

Tableau 1.5: SEUs détectés à l'Aiguille du Midi

Date (heure local France)	Chip	Adresse (hexa)	Données lues
24/06/2012 00 :41 :15	12 (0x0C)	0xB3022	0x4555 (0100010101010101)
24/06/2012 00 :41 :15	12 (0x0C)	0xA3122	0x7555 (0111010101010101)
24/06/2012 00 :41 :15	12 (0x0C)	0xA3023	0x4555 (0100010101010101)
24/06/2012 00 :41 :15	12 (0x0C)	0x73023	0x4555 (0100010101010101)
24/06/2012 00 :41 :15	12 (0x0C)	0x33122	0x4555 (0100010101010101)
24/06/2012 00 :41 :15	12 (0x0C)	0x23123	0x4555 (0100010101010101)
27/06/2012 00:54:17	62 (0x3E)	0x5F7E0	0x5551 (0101010101010001)
28/06/2012 01:22:49	40 (0x28)	0xC7340	0x55D5 (0101010111010101)
28/06/2012 19:27:15	00 (0x00)	0x2DC57	0x555D (0101010101011101)
28/06/2012 19:27:15	39 (0x27)	0x8DD56	0x5545 (0101010101000101)
28/06/2012 19:27:15	39 (0x27)	0x5DC57	0x555D (0101010101011101)
28/06/2012 19:27:15	39 (0x27)	0x4DC56	0x5545 (0101010101000101)

Comme montré dans le tableau 1.4, trois SEU, un MCU sextuple et un MCU triple ont été détectés durant cette courte période. La plateforme étant exposée en plein air, donc les énergies des

[2] Le dispositif utilisé est réalisé par des étudiants de Phelma sous la direction du Dr. Juan O'campo dans le cadre d'un projet fin d'études.

particules surtout les neutrons étaient conservés avant l'interaction avec la couche silicium des SRAMs sous-test.

La confrontation des résultats obtenus à Puno à ceux obtenus à l'Aiguille du Midi pendant la même période montre que le SER des mémoires SRAM opérant dans l'atmosphère terrestre dépend de plusieurs facteurs dont les plus importants sont:

- La latitude et la longitude.

- La technologie de fabrication des circuits dans les deux plateformes (mixte à Puno, et toute en 130nm à l'Aiguille du Midi).

- L'emplacement de la plateforme de test (à l'intérieur d'un bâtiment à Puno et en plain air l'Aiguille du Midi).

- La capacité mémoire des deux chips étant la même, la zone sensible aux SEU des mémoires en technologie 130 nm est en principe plus grande car les transistors - et donc leurs nœuds sensibles- occupent une surface physique plus importante que celle occupée par les mémoires en technologie 90nm. Ceci explique la tendance des résultats observés à l'opposée de ce que nous attendions. Cependant il est important de noter que sans avoir des informations sur le potentiel changement au niveau design/layout, la tendance concernant la sensibilité aux neutrons atmosphériques de ces deux générations des mémoires SRAM ne peut pas être généralisée à d'autres cas hors de ce cas particulier.

1.5 Conclusions

La problématique liée aux effets des radiations sur les circuits intégrés a été abordée au cœur de ce chapitre. L'environnement radiatif ainsi qu'un aperçu des différents effets dus aux radiations ont été décrits.

Les erreurs détectées dans des mémoires SRAMs, provenant de deux générations successives de technologie (90nm et 130nm), exposées aux effets de neutrons atmosphériques rencontrés à haute altitude ont été montrées, mettant en évidence la criticité potentielle de cette conjoncture pour des applications qui requière une haute sûreté de fonctionnement. Les résultats obtenus lors de l'activation aux altitudes très élevées (vols long-courrier, ballons stratosphériques et villes à hautes altitudes) de la plateforme expérimentale utilisée pour ces recherches ont été confrontés à ceux issus de la plateforme de prédiction MUSCA SEP^3/ADDICT. Une bonne corrélation entre mesures et prédictions ont été observée validant la méthodologie de ces expériences.

Les erreurs détectées (SEU, MBU et MCU) mettent en évidence une sensibilité relativement importante des mémoires SRAM testées lorsqu'elles opèrent dans l'atmosphère terrestre. Il est clair que la probabilité d'occurrence des fautes est très faible au niveau du sol, mais les très grands nombres d'applications utilisant des circuits intégrés avancés et opérant dans le même instant dans notre planète rend non négligeable l'occurrence d'erreurs qui pourraient avoir des conséquences sérieuses pour des applications critiques (transport, biomédecine …).

De plus, les résultats ont montré que les mémoires en technologie 130nm sont plus sensibles face aux soft-erreurs que celles issues d'une génération successive en technologie 90 nm. Pour faire face aux SEU, des techniques de tolérance aux fautes au niveau design et/ou layout, par exemple la technique appelée *bit interleaving* qui consiste à faire en sorte que les adjacences physiques et logiques des cellules mémoires ne soient pas les mêmes peuvent avoir été implémentées. Une telle technique expliquerait la plus faible la probabilité d'occurrence des erreurs de type MBU observées dans nos expériences à hautes altitudes dans le cas des mémoires en technologie 90 nm. Ceci dit, les progrès permanents dans les technologies de fabrication constituent un challenge significatif aux techniques de tolérance aux fautes au niveau design/layout. Une perspective des recherches réalisées dans cette thèse sera le développement et l'expérimentation d'une plateforme expérimentale à base des SRAM en technologies plus avancées, ceci pour étudier l'évolution de cette thématique.

Chapitre 2. Méthodes et outils pour l'étude des effets des radiations sur les circuits intégrés

Dans ce chapitre est donnée une introduction sur les principales méthodes et outils qui sont nécessaires pour qualifier les circuits intégrés face aux effets des radiations. Pour illustrer une méthode utilisée dans le cadre de cette thèse pour évaluer le taux d'erreurs, une campagne d'injection des fautes réalisée sur un microcontrôleur PSOC sera présentée.

2.1 Notion de section efficace

La *section efficace* (ou cross section) est une grandeur qui a pour but exprimer la sensibilité d'un composant exposé à des radiations ionisantes. Cette grandeur, notée σ, caractérise la surface sensible du composant et est exprimée en cm^2. En d'autres termes, σ donne la probabilité qu'une particule ionisante, traversant une surface de 1 cm^2, génère un événement de type SEE. Elle dépend directement de l'énergie déposée dans les matériaux, LET, suite à l'impact des particules incidentes.

La section efficace est donnée par l'équation 2.1 dans laquelle est estimé le nombre moyen des particules incidentes qui est nécessaire pour générer un événement:

$$\sigma = \frac{N_{ev}}{\emptyset} \qquad \text{Eq 2.1}$$

$\emptyset$ étant la fluence (flux intégré dans le temps) du faisceau de particules et N_{ev} étant le nombre total d'événements observés lors du test.

La caractérisation d'un circuit intégré face aux SEE est spécifiée par la courbe de section efficace en fonction du LET. Cela permet de déterminer deux paramètres caractéristiques: Le LET_{th} qui correspond à la première énergie pour laquelle des événements sont observés et la section efficace de saturation, notée σ_{sat} , qui représente la surface sensible totale du composant testé.

La figure 2.1 montre l'allure typique d'une courbe de section efficace avec ses grandeurs caractéristiques.

A partir du LET seuil (LET_{th}), et de la section efficace de saturation (σ_{sat}), il est alors possible de tracer une courbe idéale, voir équation 2.2, à l'aide de la distribution de Weibull.

$$\sigma = \sigma_{sat}\left[1 - exp\left(\left(-\frac{LET - LET_{th}}{W}\right)^{s}\right)\right] \qquad \text{Eq 2.2}$$

W étant le paramètre de largeur et S le paramètre de forme. Ils peuvent être déterminés par régression linéaire à l'aide des données obtenues lors des tests effectués sous radiation.

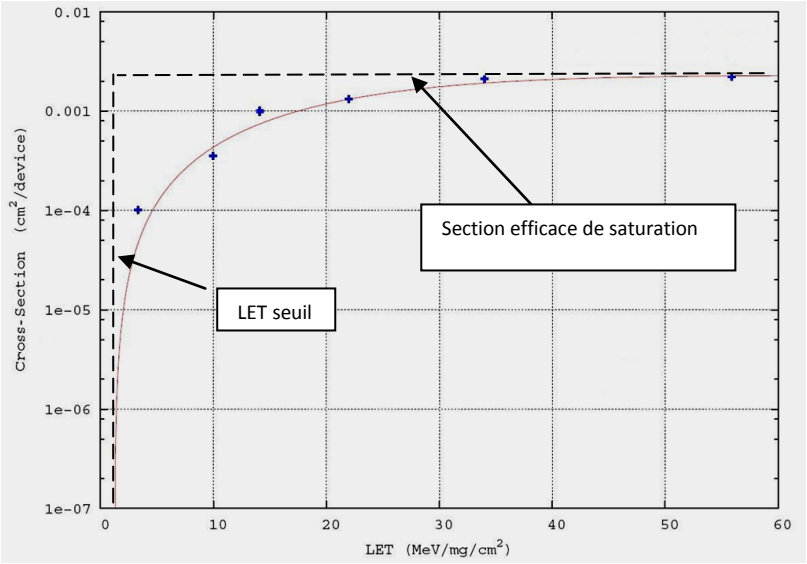


Figure 2.1: Courbe de la section efficace typique

Pour les composants de types mémoires, où le nombre de bits du circuit est connu, σ peut être exprimé en cm^2/bit ou en cm^2/composant.

On peut distinguer deux notions de la section efficace: la section efficace statique et la section efficace dynamique.

La section efficace statique d'un circuit intégré, σ_{SEU}, représente la sensibilité de ce circuit en termes de nombre de particules nécessaires pour provoquer un upset dans l'une de ces zones sensibles. Cette valeur est issue de l'exécution du test statique décrit dans la section suivante.

La section efficace dynamique d'un circuit étudié exécutant une application donnée, $\sigma_{SEU}(\text{application})$, est égale au nombre d'erreurs détectées pendant le test dynamique, décrit dans la section suivante, par le nombre total de particules auxquelles il a été exposé.

2.2 Tests statiques et dynamiques

Dans le cas des circuits digitaux complexes, deux types de tests sont utilisés pour caractériser leurs sensibilités face aux SEE: le test statique et le test dynamique. Le but est toujours l'estimation de

la sensibilité du circuit cible par rapport au nombre moyen de particules qui peuvent provoquer un mal-fonctionnement du circuit (erreur dans les résultats ou perte de séquence).

Le test statique est généralement utilisé pour caractériser la sensibilité aux SEE de circuits de types mémoires (SRAM, DRAM, mémoire 3D...) ou des blocs mémoire présents dans des circuits complexes (processeurs, FPGA, SoC, etc…). Le test statique typique consiste à écrire un motif dans la mémoire étudiée, attendre un certain temps puis lire et comparer les résultats obtenus à ceux de référence. Cette expérience est répétée pour plusieurs particules afin de déterminer la section efficace pour des LETs différents. Ce type de test nous donne une estimation « pire cas » du taux d'erreurs du circuit considéré.

A titre d'exemple, considérons une mémoire SRAM utilisée dans une architecture à base de processeur, la faute peut affecter un mot mémoire qui a déjà été utilisé par le programme exécuté, donc le SEE dans ce cas n'aura pas des conséquences. Par contre si la faute affecte un mot qui sera utilisé par une instruction prochaine, la probabilité d'avoir un résultat erroné n'est pas nulle. Enfin, si le contenu de la mémoire n'est pas complètement utilisé par le programme en cours d'exécution, la section efficace statique peut surestimer d'une manière significative le taux d'erreur dû à l'effet des particules incidentes. Ceci s'applique aussi aux processeurs plus complexes qui sont le cœur de cette thèse.

Lorsque le circuit est un processeur, le test statique consiste en l'exécution d'une série d'instructions, écriture du motif de test, attente et lecture et comparaison avec le motif de référence, permettant de cibler toutes les cellules mémoires (registres, mémoire interne) accessibles via le jeu d'instruction. Dans ce cas l'activité réalisée par le processeur est loin d'être représentative de celle de son activité typique, car ses zones mémoires ne sont pas toutes simultanément utilisées par le programme sous test. Donc, Pour obtenir des estimations de la sensibilité des circuits plus proches de la réalité, il faudrait utiliser un test dynamique au cours duquel une application proche de celle qui sera exécutée par le processeur dans son utilisation finale. Dans tout les cas, la section efficace statique des radhards (Composants durcis) même des COTS (Commercial Off-The-Shelf) devrait être fournie par le fabricant pour donner une vision claire de la sensibilité des composants surtout s'ils sont destinés à opérer dans l'espace ou à hautes altitudes.

Au moment du test sous radiation, l'application finale n'étant pas disponible durant la phase de sélection des circuits utilisés dans l'environnement spatial, par exemple, une méthode alternative doit être appliquée pour prédire d'une manière précise le taux d'erreur. Dans les références [Rezgui 2001] et [Peronnard 2009], il est démontré que l'injection de fautes par des moyens logicielles/matérielles constitue une alternative pertinente pour la prédiction des taux d'erreurs d'architecture à base de processeurs. La méthode CEU (Code Emulated Upset) est une méthode représentative d'une telle stratégie qui doit être générique et flexible sera présentée dans ce chapitre.

Le test dynamique consiste en l'exécution d'une série de commandes/instructions qui permettent d'étudier la sensibilité, des ressources sensibles du circuit, à l'effet des radiations durant une application typique. Dans le cas d'un processeur, ceci dépend strictement de la nature du programme exécuté par le DUT (Device Under Test).

A guise de conclusion, il est montré dans des travaux présentés au début des années 90s qu'il peut résulter une différence d'ordre de grandeur d'un ou deux entre le taux d'erreur mesuré suite à un test dynamique et la section efficace mesurée par un test statique.

L'utilisation de plus en plus souhaitée dans des applications spatiales des composant COTS (les solutions radhard existe mais ils ont des prix très élevés) et l'absence des standards dans le domaine tests statiques et dynamiques donnent un grand intérêt dans les communautés scientifiques liées à la thématique: *prédiction du taux d'erreurs basée sur l'émulation des fautes*.

2.3 ASTERICS: une plateforme de test générique avancée

Les cartes du commerce peuvent être utilisées pour servir comme injecteur des fautes dans des circuits cibles, mais celles-ci n'offrent pas généralement ni la flexibilité de contrôle ni l'observation des signaux et des ressources sensibles du circuit. Ceci nécessite une plateforme de test dédiée pour ce but. THESIC (Testbed for Harsh Environment Studies of Integrated Circuits), un premier exemple d'une telle plateforme flexible et générique, a été mise au point dans [Velazco 1998] [Faure 2002]. Le but de cette plateforme est d'être générique et flexible, c.à.d. permettre une interface simple avec une large palette de composants numériques (mémoires, processeurs, FPGA...) ainsi que être facilement utiliser dans le cadre des tests et d'expériences réalisées à l'aide de différentes installations utilisées dans ce domaines: accélérateurs de particules, bancs laser, expériences en environnement naturel...).

La plateforme de test ASTERICS (Advanced System for TEst under Radiation of Integrated Circuits and Systems), est une évolution majeure du testeur THESIC+ développé au laboratoire TIMA [Faure 2002].

Le testeur ASTERICS est basé autour de deux FPGAs, le premier appelé *Control FPGA*, embarque un processeur *PowerPC440* qui gère la communication entre l'ordinateur et les ressources disponibles sur la carte mère ASTERICS, et inclut un WATCHDOG programmable pour vérifier les erreurs provoquant des pertes de séquence dans des processeurs. Il surveille également la consommation du courant du DUT, afin de le protéger contre les défauts destructifs telles que les SELs (Single Event Latchups) qui peuvent survenir au cours des expériences en accélérateurs de particules. Le transfert des données est effectué via un réseau Ethernet à haute vitesse. Le deuxième FPGA, appelé *Chipset FPGA*, contient le design de l'utilisateur, qui peut être le DUT à tester, ou le

design (contrôleur mémoire, superviseur du processus de l'injection des fautes, etc.) utilisé comme interface pour les ressources de l'ASTERICS, et/ou la carte fille (SRAM, DDR2, processeur, microprocesseur...). De cette manière sont réduits au minimum le temps et l'effort de développement, et le coût de la plateforme matérielle nécessaire pour effectuer des tests sous radiations ou des expériences d'injection de fautes sur un nouveau circuit [Peronnard 2008]. Figure 2.2 donne une illustration sur l'architecture du testeur ASTERICS tandis que la figure 2.3 montre une photo de la plateforme de test.

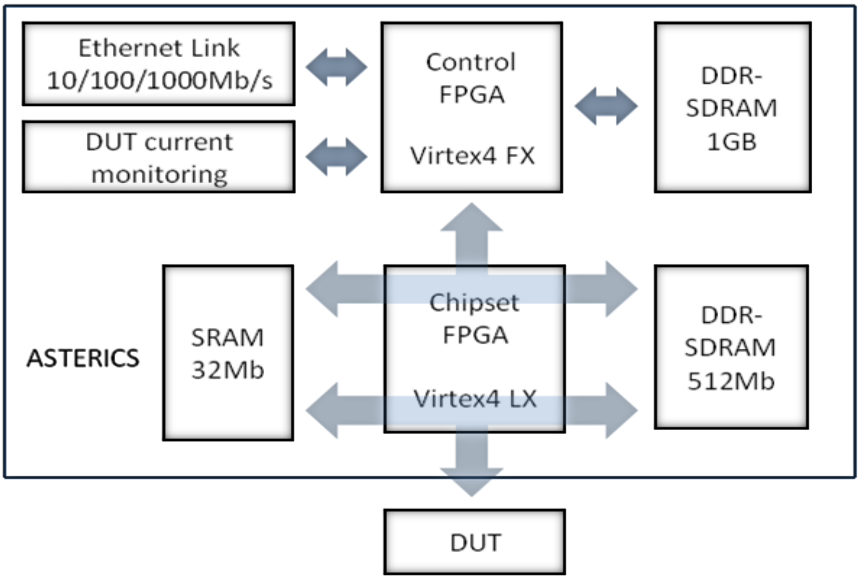


Figure 2.2: Architecture de la plateforme de test ASTERICS

Figure 2.3: Plateforme de test ASTERICS

Pour faciliter la communication avec le testeur, on utilise une application API (Application Programming Interface), qui est basée sur un ensemble des fonctions telles que la lecture et l'écriture de la mémoire, la configuration du *Chipset FPGA*, le contrôle des limites de consommation du courant du DUT, etc. Cet ensemble se trouve sous forme d'une librairie DLL (Dynamic Link Library) qui peut être utilisé dans le logiciel de l'interface avec le testeur.

Les caractéristiques importantes de ce testeur sont: la facilité d'adaptation à tout type de DUT et la possibilité de le contrôler à distance grâce à son câble Ethernet.

2.4 Moyens de tests usuels pour la caractérisation des composants intégrés face aux radiations

Plusieurs moyens peuvent être utilisés au niveau du sol pour la qualification des circuits intégrés face aux effets des radiations. Les sources radioactives, accélérateurs de particules et les faisceaux lasers sont des moyens qui permettent l'obtention des résultats beaucoup plus rapidement que les tests faits dans l'environnement réel, grâce au flux qu'ils produisent qui est bien supérieure à celui qui existe dans la nature.

2.4.1 Sources radioactives

C'est un moyen peu couteux basé sur l'émission de certains types de particules de LET différents, ce qui permet d'avoir des informations préliminaires sur la section efficace d'un composant. Deux exemples de telles sources sont : le Californium 252 et l'Américium 242. Dans le cas de californium 252, les particules émises peuvent être des particules alphas ainsi que deux types d'ions lourds ayant des LETs de 45 et 46 $MeV.cm^2/mg$ .Les faibles énergies des particules générées par ces sources les empêchent de pénétrer plus de 15 µm environ dans la matière. Elles sont donc utilisables seulement dans le cas où le composant possède une couche superficielle peu importante.

2.4.2 Accélérateurs de particules

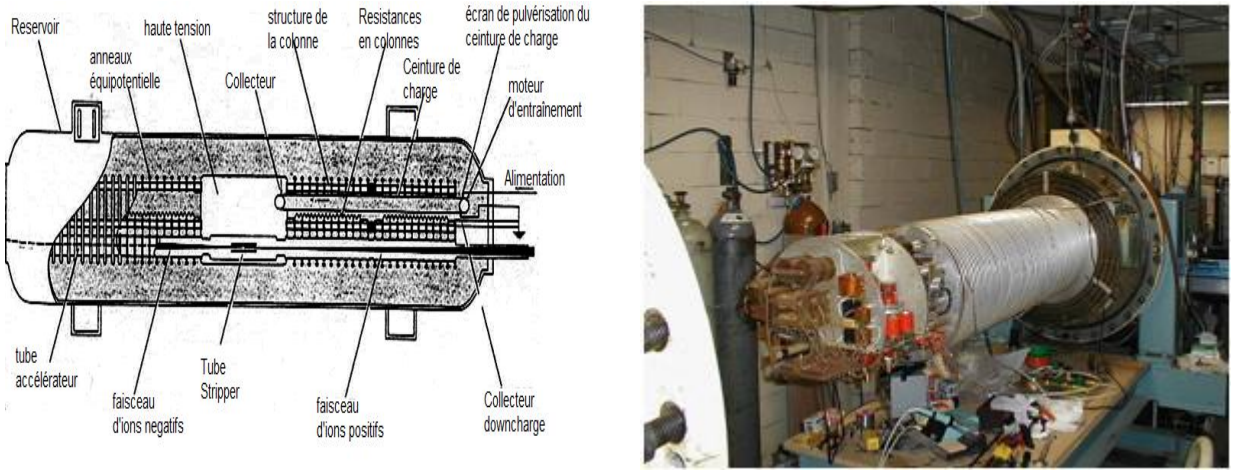
Ce sont les instruments les plus utilisés pour la détermination de la sensibilité d'un composant vis-à-vis des particules radiatives. Ils produisent des particules énergétiques telles les ions lourds, les protons, et les neutrons, puis ils les accélèrent via un champ électrique ou magnétique.

On peut distinguer deux types d'accélérateurs de particules:

- Les accélérateurs de type Tandem Van de Graaff: IPN (Orsay, France), TASSC (Canada), BNL (Upton,USA), SLAC (Stanford, USA).

- Les accélérateurs circulaires de type cyclotron: CYCLONE (UCL, Belgique), TRIUMF (Vancouver, Canada), LBL (Berkeley, USA), GANIL (France), VIVITRON (Strasbourg, France).

Les accélérateurs Tandem Van de Graaff, figure 2.4, sont des machines électrostatiques par "addition": les charges sont accumulées sur un conducteur grâce à un convoyeur mobile. Ils sont capables d'accélérer des ions lourds à une différence de potentiel d'environ 100 MV. Les charges, qui peuvent être négatives ou positives, selon la source, subissent alors l'accélération entre les deux électrodes, après changement de charge. Ces générateurs qui ont été développés à la fin des années 1920s et qui sont devenus très populaires en peu de temps, dérivent de série des machines électrostatiques du XVIIIème siècle [Brenni 1999].



(a) Schéma d'un tandem Van de Graaff.

(b) Accélérateur Van de Graaff de 2 MeV datant des années 1960 ouvert pour maintenance.

Figure 2.4: Tandem Van de Graaff

Le cyclotron, figure 2.5, est un type d'accélérateur de particules circulaire inventé par Ernest Orlando Lawrence en 1931. Les particules placées dans le champ magnétique du cyclotron suivent une trajectoire spirale et sont accélérées par un champ électrique alternatif à des énergies allant depuis quelques MeV jusqu'à 85 MeV dans le cas du cyclotron de l'UCL. Ce type d'accélérateurs permet d'obtenir un flux de plusieurs dizaines de milliers de particules par seconde, qui peuvent traverser une très bonne épaisseur de silicium. D'autres types d'accélérateurs circulaires, d'invention plus récente, permettent d'atteindre des énergies supérieures : synchrocyclotron (centaines de MeV) et synchrotron (millions de MeV, ou TeV).

2.4.3 Faisceau laser

Un faisceau laser peut être utilisé pour l'injection des SEEs dans un circuit, ceci d'une façon complémentaire à celle des accélérateurs de particules [Buchner 1987] [Richter 1987]. En effet, le faisceau laser peut toucher une partie sensible du circuit de l'ordre du micron, tandis-que les accélérateurs touchent la surface entière. A l'issu d'un test laser, il peut donc être obtenue une cartographie précise de certaines zones sensibles.

Malgré ses avantages, le test laser est soumis à deux limitations :

- La réflexion du faisceau par les couches de métallisations, ce qui pose des problèmes pour les composants complexes ayant plusieurs couches de métallisations. Ceci oblige à les attaquer de la face arrière ce qui requière des travaux d'amincissement du *packaging* et éventuellement du substrat du circuit s'il est *flip-chip* pour exposer la zone active au laser.
- L'énergie déposée par les photons lors du test laser n'a pas jusqu'à présent été corrélée au LET d'une particule énergétique [Pouget 2001].

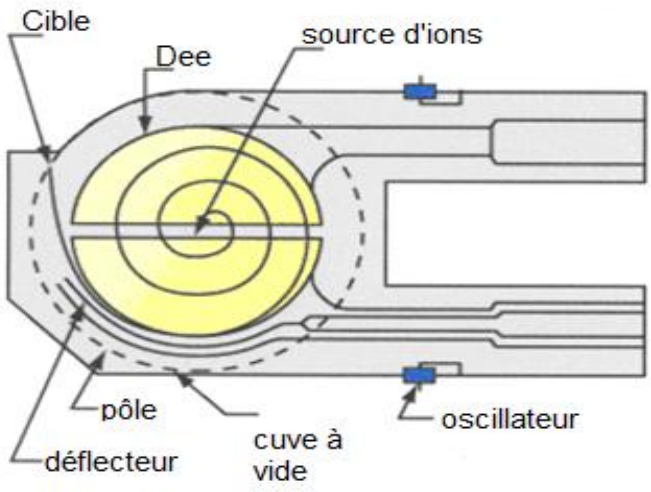


Figure 2.5: Schéma d'un Cyclotron.

2.4.4 Tests en environnement réel

Divers essais, appelés Real-Life Tests dans la littérature, peuvent être réalisés dans l'environnement naturel, dans l'atmosphère terrestre à hautes altitudes, ou dans l'espace ayant pour but d'étudier l'effet des radiations sur les circuits intégrés.

Les essais à hautes altitudes peuvent avoir lieu dans des avions, dans des montagnes, ou dans des ballons stratosphériques. Ces essais ont pour but la caractérisation du milieu pour établir ou valider des modèles [Normand 2001], ainsi que pour l'étude des effets des radiations sur les équipements et les personnels [Chee 2000] [Taber 1993] [Olsen 1993] [Goldhagen 2000] [Sohn 2000].

Dans le chapitre 1, ont été présentés les résultats d'expériences Real-Life au cœur de cette thèse qui ont eu pour but l'obtention des données objectives sur la sensibilité dans l'environnement réel des mémoires SRAMs commerciales issues de deux générations de technologies successives.

Pour ce qui concerne les essais qui prennent place dans des satellites, il est clair que les rayons cosmiques présents dans les orbites considérées offrent une opportunité unique pour obtenir un retour objectif et effectif sur l'impact des particules dont les énergies dépassent celles disponible lors des tests en accélérateurs. De plus de telles expériences permettent de valider l'immunité des circuits durcis face à l'effet des radiations présentes dans ce type d'environnement certifiant donc leurs utilisations dans les applications spatiales. On peut citer quelques projets auxquels le laboratoire TIMA a participé et qui ont donné lieu à différentes publications :

- Le projet STRV (Space Technology Research Vehicles) [Web STRV] dirigé par le DRA (*Defence Research Agency*, Royaume-Uni), lancé en juin 1994. Il est composé de deux satellites qui furent placés sur une orbite GTO (*Geostationary Transfer Orbit*). L'expérience développée au laboratoire TIMA concernait la logique floue. Suite à un incident technique survenu en orbite, Cette carte n'a pas pu être alimentée et donc de résultats n'ont pas été obtenus.
- Le projet MPTB (Micro-electronique and Photonic TestBed) [Web MPTB] qui a été dirigé par le NRL (*Naval Research Lab*) [Web NRL]. Dans la charge utile d'un satellite lancé en novembre 1997, ont été inclues 24 expériences destinées à la mesure directe des effets de radiations sur divers types de composants, ainsi qu'à l'obtention des mesures réalistes sur l'environnement radiatif spatial [Ritter 1997]. Parmi ces expériences fut choisie par NRL une développée au laboratoire TIMA concernant l'étude de la robustesse intrinsèque des réseaux de neurones artificiels face aux SEU [Cheynet 1999]. L'analyse des résultats durant deux années de vol, montre 79 upsets sur les paramètres provoquant 9 messages du processus exécuté dont 7 indiquant une baisse de performance, ceci conduit à une robustesse de 91%.
- Le projet LWS-SET (Living With a Star – Space Environment Testbed) qui a pour but d'identifier le pourquoi et comment varie le soleil, comment les autres planètes répondent à ces variations, et quels sont les effets sur l'humanité. Dans la charge utile de ce satellite sont inclues six expériences destinées à l'étude des effets des radiations. L'une d'entre elles, développée à TIMA, consiste en l'étude du comportement et de l'efficacité d'une architecture implémentant la méthode de tolérance aux fautes de l'état de l'art connue sous le non de Triple Modular Redudancy (TMR) qui consiste à répliquer trois fois une application et réaliser un vote majoritaire sur les sorties. Un TMR fut implémenté sur un FPGA Virtex-II pour une application cryptographique [Foucard 2010]. Le lancement de ce satellite devait être fait en 2010 a été reporté à l'année 2014.

Conclusion, ce type de résultats possède plusieurs difficultés logistiques: perte du satellite, délai de lancement, problèmes d'alimentations électroniques etc. Les essais dans l'environnement spatial sont les essais de plus grande durée, ils peuvent prendre plusieurs années, sont très couteux et risquent de ne pas fournir de résultats. Cependant dans le cas où tout ce passe comme attendu, les résultats obtenus peuvent être extrêmement utiles pour faire progresser différents domaines tels: technique de tolérance aux fautes, technique de durcissement et l'évolution de la sensibilité des composants électroniques en fonction de la technologie etc.

2.5 Méthodologie pour simuler l'impact des fautes SEU dans des processeurs

Une méthode, appelée CEU (Code Emulated Upset), a été développée à TIMA pour émuler les effets des radiations sur les circuits numériques de type processeurs, prédire le taux d'erreurs de ce processeur lorsqu'il exécute une application donnée, et pour valider diverses techniques de tolérances aux fautes en particulier celles appliquées au niveau software. Dans ce qui suit sera donné un exemple de l'application de la méthode CEU à un microcontrôleur PSOC [1].

2.5.1 La méthodologie CEU pour l'injection de fautes

La méthodologie CEU est une approche ayant pour but l'injection d'erreurs de type SEU dans un circuit de type processeur ou microprocesseur. L'émulation des SEU dans ce type de circuits peut être obtenue par l'exécution d'un code approprié qui peut être activé grâce à l'assertion d'un signal approprié tel les signaux d'interruptions asynchrones qui sont disponibles dans la plupart des processeurs à un instant choisi. Un exemple de l'application de la méthode CEU sur chacun des registres internes du microcontrôleur PSOC sera présenté dans la section suivante.

Par exemple, pour provoquer un changement du contenu d'un bit d'un registre d'usage général ou d'un emplacement de mémoire (interne ou externe) accessible par adressage direct, on a besoin de quelques instructions simples pour effectuer les tâches suivantes:

- Lecture du contenu de la cible qui doit être perturbée (registre ou mot de mémoire à changer).
- Opération de type OU exclusif (XOR) de la valeur lu avec une masque (xor avec un "1" à l'emplacement de bit qui doit être inversé et «0» ailleurs).
- Ecriture de la valeur corrompue dans le contenu de la cible.

Les jeux d'instruction de la plupart des processeurs permettent la réalisation de ces taches en quelques instructions pendant quelques cycles d'horloge. L'aspect crucial pour simuler le

[1] Le microcontrôleur PSOC est fourni par Cypress Semiconductor

bouleversement d'un bit est l'inclusion de ces codes dans le programme initial, pour qu'il soit exécuté d'une manière non intrusive (seul le bit cible d'une cellule mémoire sera perturbé) au moment voulu lors de l'exécution de l'application.

Une solution efficace et simple est d'utiliser le mécanisme d'interruption qui est disponible dans presque tous les processeurs. Pour ce faire, le code qui va causer la perturbation, terminé par l'instruction retour de l'interruption «*RETI*», doit être enregistré à une adresse spécifique dans la mémoire pour être utilisé en tant que programme associé à l'interruption, le code CEU dans ce cas. En effet, si le processeur est configuré d'une manière adéquate, il effectue habituellement les étapes suivantes, illustrées dans la figure 2.6, en réponse à l'activation d'une interruption.

- L'arrêt de l'exécution du programme après avoir terminé l'exécution de l'instruction en cours.
- La sauvegarde du contexte dans une pile. Cette étape dépend strictement de l'architecture du processeur étudié. Dans la plupart des processeurs, le compteur des programmes et les registres des flags seront automatiquement sauvegardés dans une pile.
- L'exécution du code CEU, provoquant ainsi l'inversion du ou des bits de la cible choisi.
- La restauration du contexte. Comme la deuxième étape, celle-ci dépend de l'architecture.

L'activation du signal d'interruption à des instants aléatoires lors de l'exécution d'un programme permettra l'injection des fautes dans les ressources accessibles par le jeu d'instruction, simulant donc d'une manière assez réaliste les effets (SEU, MCU, MBU et SEFI) dus à l'impact des particules énergétiques.

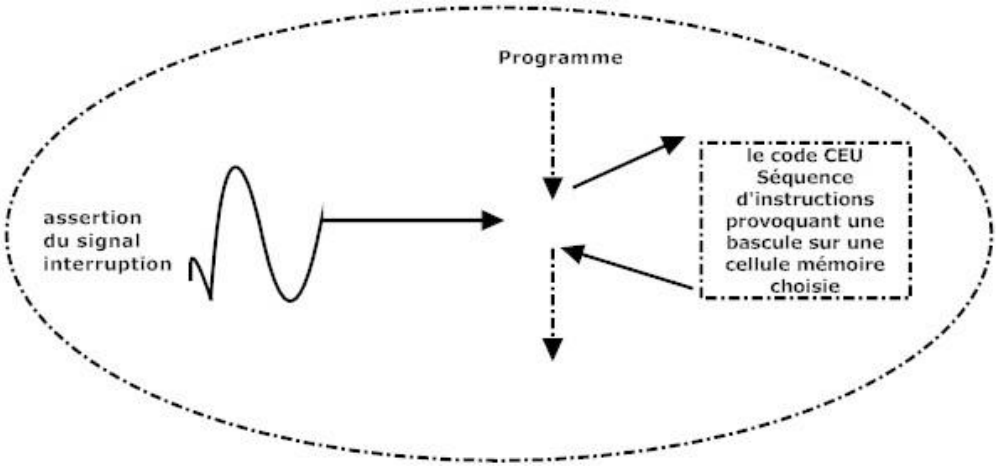


Figure 2.6: Diagramme de la méthode CEU

Le but finale de cette méthode, qui a été présentée pour la première fois en 2000 [Velazco 2000], était l'obtention à partir des données issues des campagnes d'injection de fautes, d'une

prédiction réaliste de la sensibilité face aux effets SEE considérés d'un composant de type processeur ou microprocesseur lorsqu'il exécute une application donnée.

Le taux d'erreur appelé τ_{inj} peut être dérivé du rapport entre le nombre observé d'erreurs pour une application déterminée et le nombre de SEUs injectés:

$$\tau_{inj} = \frac{\#\ Erreurs\ detectées\ par\ injection\ de\ fautes}{\#\ de\ particules\ injectées\ par\ injections\ de\ fautes} \qquad \text{Eq 2.3}$$

La section efficace statique σ_{static} d'un circuit, estime la sensibilité d'un circuit intégré en termes du nombre moyens de particules requis pour provoquer un upset.

$$\sigma_{\text{static}} = \frac{\#\ Erreurs\ detectées\ sous\ radiations}{\#\ particules\ sous\ radiations} \qquad \text{Eq 2.4}$$

La sensibilité aux SEU τ_{SEU} d'un processeur exécutant une application donnée, est estimée comme le nombre moyen des particules nécessaires pour provoquer une erreur dans le résultat du programme exécuté. D'après les équations 2.3 et 2.4 il apparait clairement que la multiplication du taux d'erreurs issu des prédictions réalisées à l'aide des techniques d'injection de fautes (τ_{inj}) par la section efficace statique σ_{static} issue des tests sous radiations correspond à la section efficace dynamique du processeur exécutant le programme étudié:

$$\tau_{SEU} = \sigma_{static} * \tau_{inj} \qquad \text{Eq 2.5}$$

La méthode CEU a été utilisée dans le passé pour estimer le taux d'erreur des différents processeurs ou microprocesseurs tels le microcontrôleur 80C51 d'Intel [Rezgui 2001] et les PowerPC 7447A et 7448 exécutant des applications réalistes telles le SCAO 2 [Peronnard 2009]. Les résultats obtenus par prédictions ont été confrontées à ceux issus des tests sous radiations. Dans les deux cas les deux résultats étaient très corréler. La marge entre les résultats issus sous faisceau de radiation et ceux prédits est donnée dans le tableau 2.1, pour deux types d'ions différents (Argon et Krypton), et dans deux cas la cache des données activée et désactivée.

Tableau 2.1: Marge entre la section efficace mesurée et prédite par CEU dans le cas du PowerPC 7448

Ion	Marge dans le cas ou la cache des données est activée	Marge dans le cas ou la cache des données est désactivée
Argon	$0.08\ 10^{-5}$	$0.12\ 10^{-6}$
Krypton	$0.07\ 10^{-5}$	$0.26\ 10^{-6}$

[2] SCAO : Système de Commande d'Attitude et d'Orbite fourni par le CNES

2.5.2 Limitations de la méthode CEU, un cas étudié : Le microcontrôleur PSOC

De nombreux systèmes, sont conçus de nos jours pour l'utilisation dans des applications industrielles et commerciales, équipés de capteurs, composants d'interface analogique, acquisition de données digitales, de contrôle et de réseau sans fil. Ces applications peuvent être mises en œuvre à l'aide des SOCs (System On Chip).

Afin d'assurer le fonctionnement fiable de ces systèmes lorsqu'ils sont utilisés dans l'espace, où leur sensibilité aux effets des radiations peut être très élevée, le phénomène SEU est considéré comme critique. Dans le cas du PSOC, la méthode CEU peut être utilisée pour modifier le contenu des cellules mémoires, des registres spéciaux et des accumulateurs, ceci à un instant choisi aléatoirement, pour pré-estimer la sensibilité face aux SEU de l'application exécutée par le circuit cible.

Pour illustrer l'utilisation et les potentielles limitations de l'approche CEU dans le cas des SOCs, nous avons choisi le microcontrôleur PSOC CY8C27643. Ce circuit est basé sur un processeur M8C (Architecture Harvard 8-bit), qui contient les registres suivants: PC (Program Counter), SP (Stack Pointer), A (Accumulator), X (Index Register), et F (Flag Register). Les figure 2.7 et 2.8 illustre l'architecture du PSOC et celle du processeur M8C.

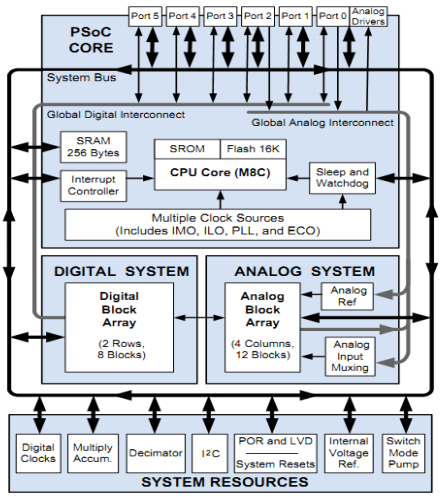


Figure 2.7: Architecture du PSOC

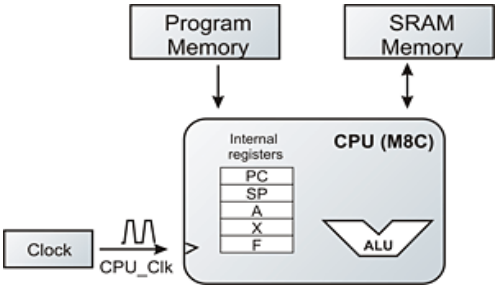


Figure 2.8: Architecture du processeur M8C

Une campagne d'injection de fautes par la méthode CEU a été réalisée sur le microcontrôleur PSOC lorsqu'il exécute une application de multiplication des matrices de dimensions 6x6. Les résultats intermédiaire ont été stockés dans 120 registres, et les sorties dans 36 registres. 8 registres

temporels ont été encore utilisés. La zone mémoire occupée par l'algorithme exécuté était 236 registres ce qui fait un pourcentage de 92.18%.

Une carte fille interfaçant le PSOC à la plateforme de test ASTERICS a été développée pour permettre la réalisation des campagnes d'injection de fautes avec la méthode CEU. Le signal d'interruption est fourni par le *Chipset FPGA* d'ASTERICS qui a été configuré pour qu'il supervise le processus d'injection de fautes. Un contrôleur mémoire a été aussi implémenté sur ce FPGA permettant l'enregistrement, dans la mémoire SRAM du testeur, des erreurs détectées lors des campagnes d'injection de fautes. Les variables tempA et tempD contiennent respectivement l'adresse du mot mémoire à perturber (address de la cible) et le masque des données data_mask (bit à perturber parmi 8-bit). Elles sont fournies au PSOC par le superviseur au début de chaque exécution. Le registre PC étant de 16 bits, il est décomposé en deux sous-registres PCH (PC low) et PCL (PC High) de 8 bits chacun. Lors de l'activation d'une interruption les registres F, PCL et PCH sont enregistrés dans une pile dont le pointeur est le registre SP. On peut accéder ces registres via les instructions *PUSH* et *POP*. Les codes CEU associés pour chacun des registres spéciaux pour l'application de la méthode CEU sont donnés par le tableau 2.2.

Tableau 2.2: Les codes effectués par la méthode CEU lors d'une interruption

A	F	X	SP	PC	Zone mémoire

xor A,tempD →A
RETI
	pop A
	xor A,tempD →A
	push A
	RETI
		xor X,tempD →X
		RETI
			pop A
			mov A→ F
			jmp opcode→ mem[add]
			pop A
			mov A→mem[add + 1]
			pop A
			mov A→mem[add + 2]
			xor SP, tempD
			jmp add
				pop A
				mov A → temp
				pop A
				mov A → PCL
				pop A
				mov A → PCH
				if(tempA == 0)
				xor PCL,tempD →PCL
				else
				xor PCH, tempD → PCH
				mov PCH → A
				push A
				mov PCL → A
				push A
				mov temp → A
				push A
				RETI
					Sauvegarder le X
					mov [tempA] → X
					mov [X] →A
					xor A,tempD →A
					mov A → [X]
					RETI
					
					[X] correspond au contenu de la zone mémoire à l'adresse X

Pour injecter une erreur dans le PC, l'instruction POP est utilisée pour l'extraction de PCH et PCL de la pile. Ensuite le code CEU sera appliqué à l'un de ces deux registres. Le choix du registre sera fait par le variable tempA qui contient l'adresse à modifier.

Le cas du SP est plus délicat. La méthode d'injection de fautes CEU étant basée sur l'activation de l'interruption, le pointeur de pile SP est une ressource clé car elle permet la sauvegarde et la restauration du contexte. Si le contenu de SP est changé par le code associé à l'interruption, la conséquence sera que le PC sera chargé en une valeur erronée avec des chances de provoquer une erreur critique telle la perte de séquence. L'injection de fautes par SP nécessite donc d'une séquence

particulière d'instructions qui permette l'accès à la pile sans utiliser SP, car SP est la cible du SEU. Cette séquence consiste à sauvegarder le code-op de l'instruction *jump* avec le contenu des adresses PCH et PCL dans une adresse de mémoire puis faire un appel *jump* après avoir changer SP par le code CEU. Bien entendu, les fautes dans SP ont lieu seulement si l'application se sert de cette ressource, ce qui n'est pas le cas dans le programme multiplication des matrices.

38700 fautes ont été injectées dans les registres du processeur ainsi que dans la zone mémoire interne lors de l'exécution du programme multiplication des matrices. Les résultats sont classifiés en trois catégories:

- Faute silencieuse: la faute injectée n'a pas d'effet sur les résultats du programme étudié.
- Erreurs des résultats : la matrice obtenue n'est pas correcte.
- Perte de séquence (timeouts): quand l'application exécutée par le PSOC ne se termine pas.

Les résultats de cette campagne d'injection de fautes sont résumés dans le tableau 2.3.

Tableau 2.3: Résultats de l'injection de fautes dans la zone sensible du PSOC

# d'injections	# erreurs	# de perte de séquence	# fautes silencieuses
38700	3471 (8.96%)	1028 (2.65%)	34201 (88.37%)

Parmi les SEUs injectés dans ces zones sensibles, 8.96% ont provoqués des erreurs. 2.65% des SEUs injectés ont résulté en perte de séquence, erreur critique car l'application nécessite un reset pour pouvoir être relancée. 88.37% des fautes injectées ont été silencieuses n'ayant donc eu aucun effet sur l'application exécutée durant ces essais.

Des campagnes d'injection des fautes ciblant une à une les ressources sensibles accessible du PSOC ont été effectuées pour identifier les ressources les plus sensibles. Ces résultats sont montrés dans le tableau 2.4.

Tableau 2.4: Résultats détaillés des campagnes d'injection de fautes

Registre	# injections	# erreurs	# timeouts
A	5000	295 (5.90%)	0 (0%)
Memoire interne	5000	406 (8.12%)	0 (0 %)
PC	5000	1814 (36.28 %)	2460 (49.20%)
X	5000	41 (0.82%)	123 (2.46%)
F	5000	5 (0.10%)	21 (0.42%)
SP	5000	0(0.00%)	0 (0.00%)

5000 fautes ont été injectées dans chacun des registres spéciaux du PSOC considéré. Les résultats obtenus montrent que la partie la plus sensible aux soft-erreurs est la mémoire interne qui occupe 97.7% de la zone sensible du microcontrôleur accessible par la méthode CEU et donc dans l'environnement réel, les fautes ont une grande probabilité d'avoir lieu dans cette ressource. 5.9% des fautes injectées dans l'accumulateur ont provoqué des erreurs dans la matrice résultats. Evidemment des fautes de type timeout n'ont pas été détectées suite de l'injection de fautes dans l'accumulateur. A l'opposée, 85,48 % des fautes injectées dans le registre PC ont eu des conséquences sur l'application (erreurs dans le résultat et timeouts). Des erreurs et des timeouts ont aussi été détectés suite à l'injection de fautes dans le registre X, utilisé dans le PSOC pour indexer la mémoire. Le registre F était le moins sensible face au soft-erreurs. Le registre SP n'a donné ni d'erreurs ni des timeouts, et ceci est justifié par le fait que dans le programme de multiplication de matrices utilisé dans cette campagne, il n'y a aucune tâche qui requière l'utilisation de la pile (comme un appel d'une fonction…).

Il est important de rappeler que la méthode CEU, ne peut pas prendre en compte les cas où la faute se produit dans les zones sensibles qui ne sont pas accessibles via le jeu d'instruction, telles les registres de contrôle des unités arithmétique et logique, les registres de pipeline et les registres des machines à états. Ceci met en évidence la nécessité d'une méthode plus générale qui considère toutes les ressources sensible et qui est applicable à tous types de circuit, pas obligatoirement les circuits de type processeurs ou microprocesseur.

Dans le cas des circuits de type processeur, dont on dispose de la version hardware, la méthode CEU est très pertinente. Cependant elle a seulement accès aux zones sensibles dont on peut accéder avec le jeu d'instructions et donc l'estimation finale du taux d'erreurs avec la méthode CEU peut ne pas être précise si la zone non accessible est importante. Le PSOC illustre bien cette situation, dans la figure 2.7 on peut clairement voir des blocs non accessibles via le jeu d'instruction tel que les blocs digitaux, les ressources du système, les ressources des horloges etc… Des tests sous radiations prévus dans le cadre de cette thèse n'ont pas pu avoir lieu pour des raisons logistiques. Donc, l'impact de cette faiblesse potentielle de la méthode CEU n'a pas pu être quantifié car la section efficace statique du PSOC n'était pas disponible. Cependant la prédiction du taux d'erreur d'une application, basée sur la multiplication de la section efficace statique par le taux d'erreur issus des campagnes d'injection de fautes reste applicable dans le cas d'injection de fautes réalisé sur un modèle HDL(Hardware Description Language) du circuit. L'extrapolation de la méthode de prédiction du taux d'erreurs dues aux SEU en utilisant des modèles HDL du circuit fera l'objet du chapitre suivant. Contrairement à la méthode CEU, qui se focalise sur des circuits de type processeur, la méthode qui sera explorée dans ce qui suit est générique mais elle exige la disponibilité d'un modèle RTL (Registre

Transfer Level) du circuit. L'une des contributions importante de cette étude sera de fournir des éléments sur la sensibilité du circuit étudié avant sa fabrication et identifier les *Talons d'Achilles* des stratégies de tolérance aux fautes adoptées offrant donc un feedback pour y faire face et pour améliorer la robustesse du circuit.

2.6 Conclusions

Dans ce chapitre ont été décrits les méthodes et outils qui peuvent être utilisés pour l'évaluation de la sensibilité des circuits intégrés faces aux évènements singuliers provoqués par l'impact, dans des zones sensibles, des particules énergétiques présentes dans l'environnement dans lequel ils opèrent.

Une méthode d'injection de fautes et de prédiction de taux d'erreurs, la méthode CEU développée à TIMA, a été décrite et appliquée à un microprocesseur PSOC. Les avantages ainsi que les limitations de cette méthode ont été montrées pour mettre en évidence le besoin d'une nouvelle méthode d'injection de fautes. Dans le chapitre suivant, la nouvelle méthode sera décrite et appliquée a trois circuits différents : un processeur complexe, le LEON2, un microcontrôleur 80C51 d'Intel et enfin un réseau de neurones artificiels de type Hopfield.

Chapitre 3. Emulation de fautes au niveau netlist: La méthode NETFI

Dans ce chapitre est présentée la méthode étudiée dans le cadre de cette thèse pour émuler les conséquences des radiations sur des circuits intégrés dont on dispose du modèle HDL (Hardware Description Language). Le but est d'émuler les erreurs de type SEU, SETs dans la zone sensible des circuits intégrés digitaux. Des erreurs de types Stuck_at, utilisées pour identifier un défaut de fabrication dans le circuit considéré, peuvent être aussi injectées. Cette méthode qui fut appelée NETFI (NETlist Fault Injection) est générique et automatisable. Elle sera illustrée en détail dans le cas de son application à un processeur LEON2. Les buts principaux de l'injection de fautes sont la prédiction du taux d'erreurs et la validation des stratégies de tolérances aux fautes implémentées à différents niveaux. L'application de NETFI à un microcontrôleur 80C51 d'Intel permettra de comparer les résultats issus de la prédiction du taux d'erreurs à ceux issus de tests sous radiations. NETFI sera aussi appliqué à deux versions d'un réseau de neurones de type Hopfield, une standard et une durcie au niveau du design, développées dans le cadre de cette thèse pour valider les techniques de tolérance aux fautes implémentées.

3.1 Etat de l'art de l'injection de fautes

Dans les dernières années, l'étude de la sensibilité des circuits par injection des faute est devenue une technique très populaire pour la détermination expérimentale des paramètres de fiabilité d'un système par rapport aux effets des radiations [Shokrallah 2008].

Parmi les nombreuses approches d'injection de fautes qui ont été proposées, il y a deux catégories [Folkesson 1998] :

1- Injection de fautes basée sur le matériel ou HWIFI (Hardware-Implemented Fault-Injection) [Madeira 1994][Arlat 1990].
2- Injection de fautes à base logiciel: Ces méthodes peuvent être divisées en deux classes, les logiciels qui implémentent l'injection de fautes ou SWIFI (Software-Implemented Fault-Injection), et les injections de fautes basées sur la simulation (Spice, modelsim, etc..).

Il existe aussi une catégorie basée sur la combinaison du matériel et du logiciel, ce type d'injections de fautes est appelé *Hybrid fault-injection*.

Dans l'injection de fautes basée sur la simulation, les erreurs sont injectées dans les modèles HDL des circuits en utilisant les langages VHDL [Sieh 1997] [Jenn 1994] [Delong 1996] [Bou 1998], ou Verilog [Zarandi1 2003] [Zarandi2 2003]. Les principaux avantages de ce type d'injection de fautes sont sa bonne observabilité et contrôlabilité, qui permet d'accéder à la totalité des ressources

sensibles, ce qui aboutit à des résultats plus précis, et la courte durée des expérimentations d'injections de fautes car le système cible peut tourner à sa vitesse maximale.

Un autre avantage de ces techniques est la possibilité d'injection de fautes de type SEU dans les cellules mémoires ainsi que des SET (Single Event Transient) dans la logique combinatoire, chose qui n'est pas possible dans la méthode CEU car elle ne dispose pas du modèle du cas.

Dans la littérature du domaine peuvent être trouvés plusieurs travaux sur l'injection de fautes dans des circuits, en particulier dans des processeurs, implémentés sur des FPGAs. Pour la plupart des processeurs modernes les designers et les fabricants disposent du code HDL, ce qui pourrait rendre ces méthodes utilisables tôt dans le flot de conception d'un circuit pour l'estimation du taux d'erreurs dû aux radiations et de l'efficacité des techniques de tolérance aux fautes potentiellement implémentées.

La plupart des techniques d'émulation de fautes dans des circuits implémentés dans des FPGAs, injectent des fautes permanentes ou transitoires dans le modèle VHDL synthétisable du système. Malgré que le langage Verilog est très répandu, les outils d'injection des fautes pour ce type de HDL sont très rares.

Parmi les outils d'injection de fautes basés sur la simulation peuvent être mentionnés :

- VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) [Sieh 1997] : utilise une extension du langage VHDL, qui requière la modification du langage VHDL, pour injecter les erreurs dans un circuit, ce qui permet aux fabricants, qui fournissent la librairie du design, à exprimer leur connaissance du comportement des fautes injectées dans leurs composants. Cet outil utilise multithreading (lancement de plusieurs expérimentations en parallèle dans un seul processeur) dans leur expérimentation pour accélérer le processus d'injection de fautes.

- MEFISTO-C (Multi-level Error/Fault Injection Simulation TOol) [Folkesson 1998] : est un outil d'injection de fautes, basé sur le modèle VHDL du circuit. Il utilise un simulateur pour injecter de fautes via des commandes du simulateur dans les variables et les signaux du modèle VHDL.

- HEARTLESS [Rousselle 2001] : est un simulateur des fautes permanentes ou transitoires dans un modèle RTL hiérarchique (composé des modules et des sous-modules). Il est développé pour simuler le comportement d'une faute dans des systèmes séquentiels complexes tels les processeurs. Il sert également à la validation *online* des unités de test pour des processeurs embarqués.

Les outils les plus populaires d'injection de fautes basés sur l'émulation ou la SEmulation (simulation/ émulation ou *Hybrid fault-injection*) sont :

- FITO (Fpga-based fault Injection TOol): Des fautes permanentes et transitoires sont injectées dans les bascules, et les portes logiques du design étudié. Le processus d'injection de fautes

se base sur l'ajout de portes et connexions supplémentaires aux flip-flops du design initial. Son application à un processeur OpenRisc1200 exécutant les benchmarks multiplication des matrices et *bubble sort* a montré que FITO était 79 fois plus rapide que les outils d'injection de fautes basé sur la simulation [Shokrallah 2008].

- FIDYCO (Flexible on-chip fault Injector for run-time DependabilitY validation with target specific COmmand language) [Rahbaran 2004] est une combinaison matérielle/logicielle pour l'injection de fautes dont le hardware est implémenté en VHDL dans le FPGA, tandis que le software est dans le PC de l'utilisateur. L'objectif principal est le développement d'un système souple et ouvert qui est susceptible de tester divers types de composants. Pour des designs très complexes, la taille du FPGA reste la seule restriction de FIDYCO. Des résultats obtenus par FIDYCO ne sont pas disponible dans la littérature.

- FLIPPER [Alderighi 2010] : est une plateforme d'émulation de SEUs ayant lieu dans la mémoire de configuration des FPGA, basée sur la reconfiguration partielle. FLIPPER a été appliquée pour étudier l'efficacité de l'option XTMR [1] qui consiste à implémenter dans la mémoire de configuration du FPGA Xilinx une version redondante (tripliquation + vote) d'une application spatiale. Les résultats de l'injection de fautes montrent que l'utilisation d'un XTMR améliore d'une manière considérable la tolérance aux fautes dues aux SEU.

- SCFIT (Shadow Components-based Fault Injection Technique): est un outil d'injection de fautes basée sur la *SEmulation*. Il utilise des scripts TCL pour avoir accès aux différentes ressources du FPGA Altera via un câble JTAG. SCFIT a été appliqué à un processeur complexe, le LEON2, montrant que la durée de l'injection de fautes par cette méthode est sensiblement plus rapide que celle réalisée par simulation. Pour une seule injection sur un benchmark appelé MiBench [Guthaus 2001] (*BitCount*, *BasicMath* et *Qsort*), la simulation prend 3168.6 minutes tandis que dans le cas de SCFIT, elle prend en moyenne 0.3 minutes [Mohammadi 2012].

- FuSE (Fault injection Using SEmulation *(Simulation/Emulation)*) [Jeitler 2009] : est un outil d'injection de fautes qui combine la performance d'un prototype implémenté en hardware et la flexibilité ainsi que la visibilité du standard de simulation HDL. Comme FuSE est basé sur ce qu'on appelle SEmulator (Simulator/Emulator), les fautes peuvent être injectées par le simulateur dans les signaux et les variables du code RTL, soit dans un modèle HDL simulé, soit dans la netlist implémentée sur un FPGA pour accélérer le processus d'injection de fautes. FuSE a été appliqué sur un processeur SPEAR2 [2] [Web tuwien] pour montrer les temps de simulation pour plusieurs options de FuSE: simulation seule, co-simulation c.à.d. avec l'interférence du hardware, et émulation seule qui

[1] XTMR est une méthodologie TMR (Triple Modular Redundancy) développée par Xilinx.
[2] Scalable Processor for Embedded Applications in Real-time environments

est utilisée pour accélérer le processus d'injection de fautes. Les résultats obtenus ont montrés que la troisième option était environ 4000 fois plus rapide.

- FT-UNSHADES (Fault Tolerant- UNiversity of Sevilla Hardware Debugging System) présenté en détail dans [Aguirre 2005] et mise à jour dans [Miranda 2008] : est un émulateur d'injection de fautes, dans les FPGA en utilisant une technique de reconfiguration partielle. FT-UNSHADES injecte de fautes par la technique lecture-modification-écriture des bits de configuration. FT-UNSHADES a été utilisé pour effectuer des campagnes d'injections de fautes sur trois version du processeur LEON2 (le LEON2 original, La version XTMR du LEON2 et la version tolérante aux fautes FT-LEON2 [3]) dans [Aguirre 2007]. Ces campagnes ont été réalisées pour identifier les modules les plus sensibles du processeur LEON2. Les résultats on montrés que la module de *reset* était la plus sensible face aux erreurs SEU. Les injections de fautes dans les processeurs durcis n'ont donnés aucune faute.

- FIFA (Fault-Injection Fault Analysis tool) [Naviner 2011], est un outil qui utilise des saboteurs pour injecter des fautes de types SEU, MCU et stuck-at au niveau RTL du circuit. Deux versions du circuit cible sont implémentées dans un FPGA. Les synthèses montrent que cette méthode est plus robuste et performant qu'autres méthodes de l'état de l'art.

Les méthodes qui consistent en la reconfiguration du FPGA pour injecter des fautes offrent une bonne contrôlabilité/observabilité du système cible mais elles sont très limitées en vitesse car pour l'injection d'une faute, elles requièrent la configuration du FPGA ce qui peut prendre un temps considérable [Antoni 2000].

La simulation des erreurs de type SET, SEU et Stuck_at, en utilisant une gate-level netlist est la seule méthode qui donne un feedback très précis du taux d'erreurs d'un circuit. Puisque nous pouvons utiliser une représentation détaillée, par-cellule, du circuit contenant des informations structurelles accompagnées de données concernant le timing, la propagation de la faute dans la netlist est représentative de celle de la faute ayant lieu lors du fonctionnement du circuit dans son environnement réel [Nicolaidis 2010]. Malgré que l'injection de fautes par simulation donne des résultats très représentatifs du taux d'erreurs du circuit considéré, elle reste un consommateur de temps. C'est le cas de l'injection de fautes réalisées par la méthode SCFIT, qui pour une seule injection la simulation prend environ 3160 minutes, environ deux jours [Mohammadi 2012].

L'injection de fautes par émulation, au niveau netlist sur FPGA, est rarement adressée dans la littérature. Ce type d'injection de fautes est toujours soumis à des restrictions, dont la plus importante est la taille du FPGA. Dans [Civera 2001], les fautes sont injectées au niveau netlist par modification des flip-flops en ajoutant du *mask-chain* et quelques circuits combinatoires. [Zheng 2008], présente

[3] Cette version est fournie par ESA (European Space Agency)

une méthode d'émulation de fautes au niveau netlist appelée FITVS (Fault-Injection Tool for Validating SEE). Cette méthode est basée sur la manipulation de la *gate-level netlist* qui est basée seulement sur les portes logiques basiques, tel le AND, OR etc. (une bascule, par exemple, a été remplacée par 8 portes logiques de type NAND, et deux inverseurs). Un programme développé en C-Sharp (C#), appelé *Injection Manager*, responsable de la manipulation de la netlist et du contrôle de l'injection de fautes d'une manière automatisable, a été installé dans le PC de l'utilisateur. Le circuit original a été implémenté dans le FPGA en deux circuits, un circuit original (*Fault-Free*) et un autre permettant l'injection de fautes (*Faulty*). Ces deux circuits ont pour but de mapper l'un à l'autre pour savoir le nombre des nœuds, ainsi que le nœud qui a provoqué une erreur. Un module, appelé *Emulation Controller,* était aussi implémenté sur le FPGA, pour permettre l'envoie des entrées aux deux circuits (le *Fault-Free* et le *Faulty*), l'injection de fautes dans le circuit cible et la réception des sorties. Les résultats issus de l'utilisation de cette méthode sur plusieurs circuits benchmarks implémentée, sur une démo carte qui contient un FPGA Xilinx XCV2000E et deux SRAM pour sauvegarder les sorties du circuit cible, ont montrés une grande vitesse d'émulation de fautes (environ 1µs/faute). L'un des importants désavantages de cette méthode était sa consommation du hardware. En fait, il a été montré que l'implémentation de cette méthode consomme 18 fois plus des LUT (Look-Up Table) et 25 fois plus de Flip-Flop que le design original. Parmi les autres avantages on peut citer : les blocs mémoires existant dans le circuit cible ne sont pas adressés par l'injection de fautes, et l'émulation se fait *offline,* c.à.d. une supervision humaines du processus d'injection de fautes, pour le débogage si nécessaire, n'est pas possible.

Dans [Lopez-Ongil 2007] deux méthodes d'injections de fautes ont été présenté : la première via manipulation du circuit cible tandis que la deuxième utilise la reconfiguration partielle. Dans la première méthode, une duplication de la partie séquentielle devrait être mise en œuvre, afin de comparer le circuit modifié avec le circuit original. Par conséquent, le principal inconvénient de cette approche est le surcoût en surface, mais le principal avantage est l'injection de fautes accéléré. Dans l'émulation basée sur la reconfiguration partielle, deux instances du même dispositif sous test (DUT) sont mises en œuvre, une originale et une modifiée. Bit-flips ont été fournis par la reconfiguration partielle. Le principal inconvénient de cette approche est sa vitesse relativement lente.

AMUSE (Autonomous Multilevel emulation based fault-injection for soft-error evaluation) [Entrena 2012], est un outil qui intègre à la fois les niveaux RTL et netlist pour injecter des fautes de types SET. L'injection de fautes est effectuée au niveau netlist, tandis que la faute se propage au niveau RTL pour une exécution plus rapide.

La contrôlabilité et l'observabilité complète disponible par émulation au niveau netlist ainsi que l'émulation de fautes accélérée sur toute la zone sensible incluant les blocs mémoires, les flips-flops et les LUTs, d'une manière automatisable n'existe pas encore. Une méthode qui peut injecter de

fautes sans aucune restriction sur la complexité des circuits ni sur la taille du FPGA n'est pas à nos jours adressé. Une interaction hardware/software synchronisée, peut conduire à la réalisation d'une méthode d'émulation de fautes complètement automatisable au niveau netlist sur un FPGA. Cette méthode permet l'injection de fautes dans toutes les cellules mémoires du circuit considéré. La plateforme ASTERICS sera utilisée pour effectuer les campagnes d'injection de fautes d'après cette méthode. Cette plateforme permet la réalisation de ces expériences dans de temps acceptables. A titre d'exemple, une campagne d'injection de 100000 SEU, sur un microcontrôleur simple exécutant un programme benchmark (multiplication des matrices) prend environ 1 jour. Pour le cas des circuits moins complexes, tels les Réseaux de Neurones Artificiels de type Hopfield, on peut injecter 5 fautes/seconde.

Dans ce chapitre est présentée la nouvelle méthode d'injection de fautes, étudiée dans le cadre de cette thèse pour émuler les conséquences des fautes dues aux SEUs, SETs et Stuck_at dans des circuits dont on dispose du modele HDL. Cette nouvelle méthode est basée sur la modification des composants built-in de la netlist pour pouvoir injecter des erreurs en un seul cycle d'horloge dans le circuit cible.

3.2 Description de la méthode NETFI

La méthode NETFI consiste à injecter des fautes au niveau netlist d'un circuit cible dont on dispose de son HDL. Les contributions majeures de cette méthode sont l'émulation de l'impact de différents types de fautes (SEU, SET et Stuck-at) injectées dans un circuit tournant à sa vitesse nominale et l'automatisation des injections de fautes sur n'importe quel circuit (VHDL ou Verilog) sans avoir des restrictions ni sur la complexité du circuit cible ni sur la taille du FPGA dans lequel le circuit sera émulé. Cette nouvelle méthode est basée sur une interaction software/hardware mais la partie logicielle est seulement utilisée sur l'ordinateur de l'utilisateur, dans lequel sera exécuté l'outil responsable du control des campagnes d'injection de fautes. La partie hardware est implémentée sur le testeur ASTERICS, qui reçoit les commandes du PC, et supervise le processus d'injection de fautes.

La méthode NETFI est basée sur la manipulation de la netlist en remplaçant les cellules sensibles de la bibliothèque "built-in" de Xilinx par d'autres cellules ayant le même fonctionnement mais permettant l'émulation des fautes considérées.

La figure 3.1 montre un diagramme les différentes étapes de la méthode NETFI. Le code HDL du circuit cible est synthétisé par l'outil «Synplify Pro» de «Synplicity», en exécutant un script TCL (Tool Command Language). Ce script contient toutes les options de synthèse telles la génération d'une netlist Verilog, le choix du FPGA de synthèse, etc. Le *Chipset FPGA* « *Virtex-IV XC4VLX40* » du

testeur ASTERICS a été choisi comme FPGA de synthèse pour générer la netlist du circuit cible, ceci avant la modification NETFI. Pour éviter l'utilisation puis la modification des librairies complexes dans la netlist tel que les modules DSP48 (Digital Signal Processing), on peut utiliser des FPGA de générations précédentes tels le *Virtex*. La netlist obtenue après synthèse peut contenir divers types des modules built-in de Xilinx tels que les flip-flops : FD (D flip-flop), FDC (D flip-flop avec un signal clear ou reset), FDE (D flip-flop avec le signal enable), les mémoires BRAMs (blocs ram) et les circuits combinatoires mappés dans les tableaux LUTs (look-up tables), etc. Tous ces composants ont été modifiés en ajoutant un signal injection "*INJ*" qui permet d'injecter des fautes à un instant aléatoirement choisi. L'ANNEXE A montre le code Verilog des modifications effectuées sur quelques composants de la bibliothèque "built-in" de Xilinx pour permettre l'injection de fautes.

Pour ce faire, on a besoin d'un outil permettant la modification de la netlist générée après synthèse du code RTL. Cet outil, appelé MODNET (MODify NETlist) a été développé en langage C-Sharp (C#) et est présenté en détails dans l'ANNEXE B. MODNET permet le traitement d'une netlist Verilog hiérarchique (non-aplatie) générée par «Synplify Pro» qui a (.vm) comme extension. Dans le cas de netlists générées par d'autres outils de synthèse tels Xilinx ISE, Design Compiler, etc., l'outil MODNET peut être mise-à-jour pour qu'il soit capable de les traiter.

La netlist fournie par MODNET constitue à son tour un sous-bloc du circuit RTL dédié pour la configuration finale du *Chipset FPGA* du testeur ASTERICS. Ce circuit contient un contrôleur de mémoire, et un superviseur qui contrôle le processus d'injection de fautes. NETFI est responsable de refaire la synthèse de ce circuit puis de lancer l'étape *placement et routage* (post & route) afin de générer le fichier de programmation « .bit » et de configurer le *Chipset FPGA* du testeur. Une application développée en langage « C » est responsable, après la programmation du *Chipset FPGA*, de lancer une première exécution appelée « *Golden Run* » pour obtenir les paramètres des données nécessaires aux tests telles les résultats de référence et la durée d'exécution.

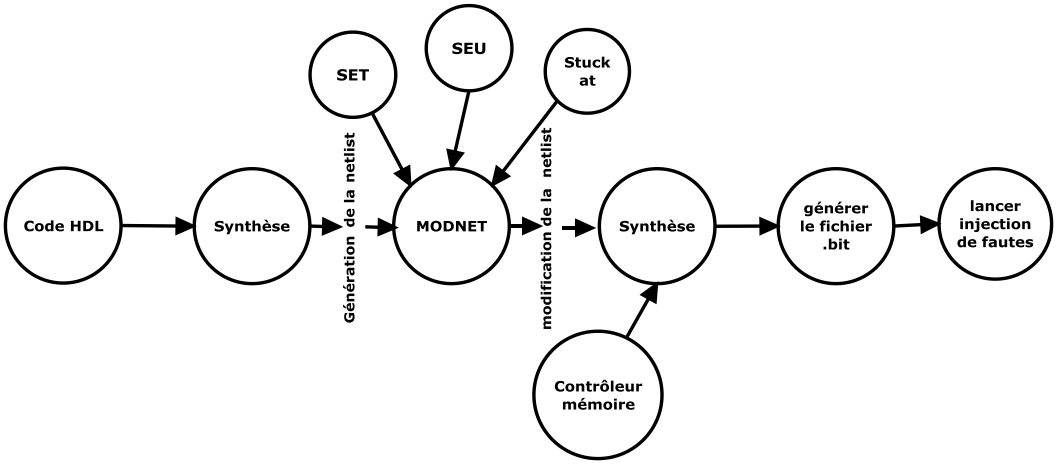


Figure 3.1 diagramme de la méthode NETFI

La capacité des FPGAs Xilinx étant limitée à un certain nombre des slices, flips-flops, BRAMs, et LUTs, quelques modifications au niveau de l'architecture de NETFI sont ajoutées pour pouvoir appliquer la méthode à des circuits complexes tels que les processeurs.

Dans le cas de microcontrôleurs ou des circuits possédant un nombre relativement faible des cellules sensibles considérées par le modèle de fautes (SEU, SET et Stuck_at) choisi par l'utilisateur, la méthode peut s'appliquer facilement. Dans les autres cas, après l'étape de la modification du netlist, MODNET permet le partage de la netlist en plusieurs sous-netlists. Ceci permet l'injection de fautes dans plusieurs parties du circuit cible. En plus, le temps nécessaire pour injecter un grand nombre de fautes dans un circuit reste inaltérable si l'option de décomposition de la netlist est choisie, ceci car le temps nécessaire pour la reconfiguration du FPGA (quelques secondes) est négligeable par rapport à la durée d'une campagne d'injection de fautes (2 à 3 jours dans le cas d'un processeur complexe tel le LEON2). La figure 3.2 montre le diagramme du NETFI dans le cas où la netlist est décomposée en deux sous-netlists.

Le temps nécessaire pour faire une campagne d'injection de fautes sur une zone quelconque du processeur est suffisant pour la génération en parallèle d'un ou plusieurs fichiers programmables ".bit". Comme conséquence de la décomposition de la netlist on peut citer :

- Même temps d'exécution puisque les sous-étapes de la décomposition seront faites en parallèle.
- Utilisation intense du processeur du PC de l'utilisateur qui est responsable du contrôle du processus de l'injection de fautes.

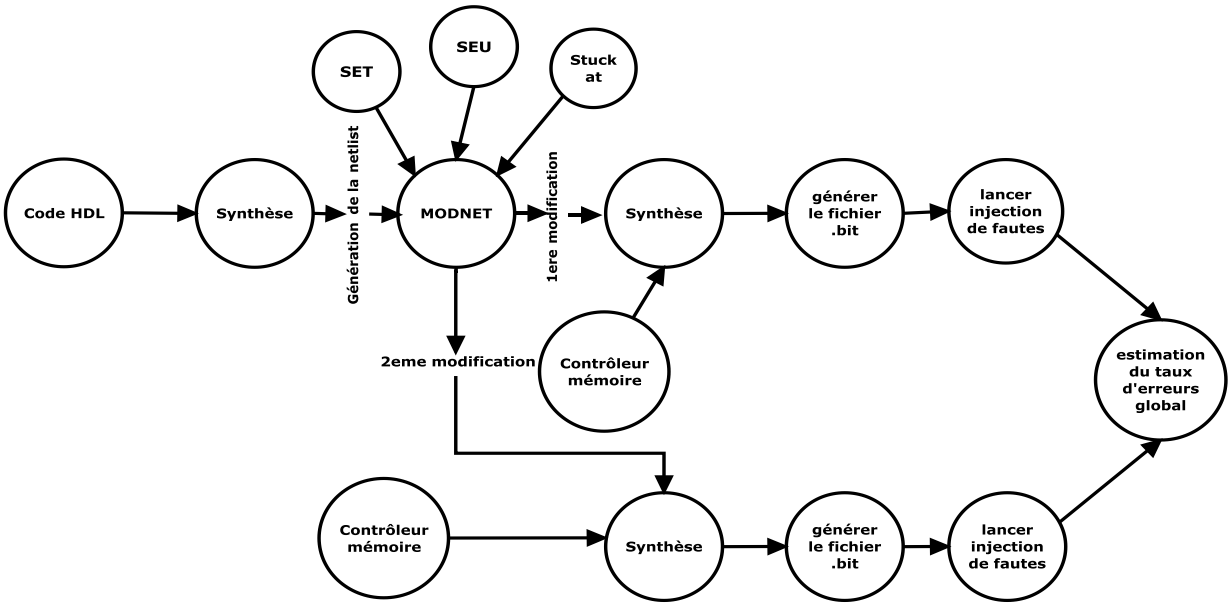


Figure 3.2 : Diagramme avancé de la méthode NETFI

L'outil NETFI est configurable et a pour entrée un fichier RTL décrit en VHDL ou Verilog, testé dans l'environnement d'ASTERICS, c.à.d. que le fichier RTL peut être implémenté comme un sous-module du circuit de configuration du testeur. La sortie de NETFI est un fichier qui contient les résultats issus des campagnes d'injections de fautes. Ces résultats peuvent être le taux d'erreurs, du circuit cible, issus de ces campagnes, l'analyse des zones les plus sensible etc. La configuration permet le choix du type des fautes parmi les fautes disponibles: SEU, SET, Stuck_at_0 et Stuck_at_1. De plus, l'option de décomposition de la netlist peut être activée dans la configuration de la méthode ou, si nécessaire, être choisie automatiquement par l'outil MODNET ce qui est le cas dans les tests des circuits complexes. L'accès à l'environnement ASTERICS permet le débogage, si nécessaire, du code RTL. L'automatisation de la méthode sous la plateforme Windows est faite à l'aide d'une application (*console prompt*) qui effectue des appels des différentes commandes :

- Tclscr.tcl : c'est un script TCL qui contient toutes les options de synthèse utilisées par « Synplify ».
- Makeclean_syn.bat : Pour nettoyer le contenu du dossier qui contient les fichiers de synthèse avant le lancement d'une nouvelle synthèse.
- Startproj.bat : Pour lancer le logiciel « Synplify » et exécuter le script Tclscrt.tcl.
- Endproj.bat : Après détection de la fin de synthèse, cette commande termine le processus du logiciel « Synplify ».
- Makeclean_bit.bat : Pour nettoyer le dossier contenant les fichiers de programmation (.bit) précédents.
- Makebit.bat : Responsable de faire l'étape placement et routage après synthèse. Il est aussi responsable de générer le fichier de programmation (.bit).
- Program.bat : Responsable de la programmation du *Chipset FPGA* du testeur ASTERICS.
- GoldenRun.bat : Pour obtenir les résultats de référence et la durée d'exécution (ce fichier est standard, mais il dépend strictement de l'application exécutée par le circuit cible donc une option de son configuration est disponible).
- LancerAPI.bat : Responsable de lancer l'application pour commencer l'injection de fautes. (ce fichier est standard, mais il dépend strictement de l'application exécutée par le circuit cible donc une option de son configuration est disponible).

Ces commandes ont été toutes implémentées dans le cadre des travaux faits dans cette thèse. Quelques unes ont une *background* Linux, qui requière l'installation des environnements *Linux-Like* sous la platforme Windows, tel que l'outil *Cygwin* [Web cygwin]. L'implémentation d'une application GUI (Graphical User Interface) peut être faite pour rendre la méthode plus facile à utiliser, ceci est prévu dans l'avenir proche. Une autre perspective sera la mise-à-jour de NETFI pour la rendre compatible à d'autres plateformes telles le Linux et le MAC.

Comme dit précédemment pour appliquer la méthode NETFI, la bibliothèque « *built-in* » de Xilinx doit être modifiée. Dans cette étape, on cible trois catégories principales de composants: les flips-flops, Les BRAMs et les LUTs. Il existe encore d'autres catégories de circuits séquentiels tels que les *shift registers* (SRL16, SRL16_1 etc.) et les blocs rams qui ont un bus de données de 1-bits (RAM16X1D, RAM32X1D, etc.). Ces composants seront modifiés pour permettre l'injection des fautes bit-à-bit comme dans le cas des flip-flops. D'autres circuits combinatoires, comme les portes logiques simples (xor2, xor3, and2, and3, etc.), seront modifiées de la même manière que les LUTs. Les codes Verilog permettant leurs modifications seront présentés dans l'ANNEXE A.

3.2.1 Modifications des flips-flops

L'émulation au niveau netlist des SEUs ayant lieu dans un circuit nécessite la modification de toutes les flips-flops, ceci pour pouvoir changer leurs contenues à un instant aléatoire. Pour ce but, on a catégorisé les flips-flops en deux types selon la disponibilité ou pas de signaux d'activation « *enable* ».

A. Les flips-flops sans signal d'activation: Ce genre de flip-flops est caractérisé par son fonctionnement continu, il a toujours une donnée active sur sa sortie. Le signal injection « *INJ*» est utilisé pour perturber le contenu de ces flip-flops pendant un seul cycle d'horloge. La figure 3.3 montre la modification appliquée sur ce genre des flip-flops. Dans Cette figure, « *D* » est le signal de donnée entrant, «*C* » est l'horloge, « *Q* » la donnée sortante et «*INJ* » est le signal qui permet l'injection de fautes en choisissant d'écrire dans la flip-flop considérée soit la donnée entrante soit la donnée entrante inversée.

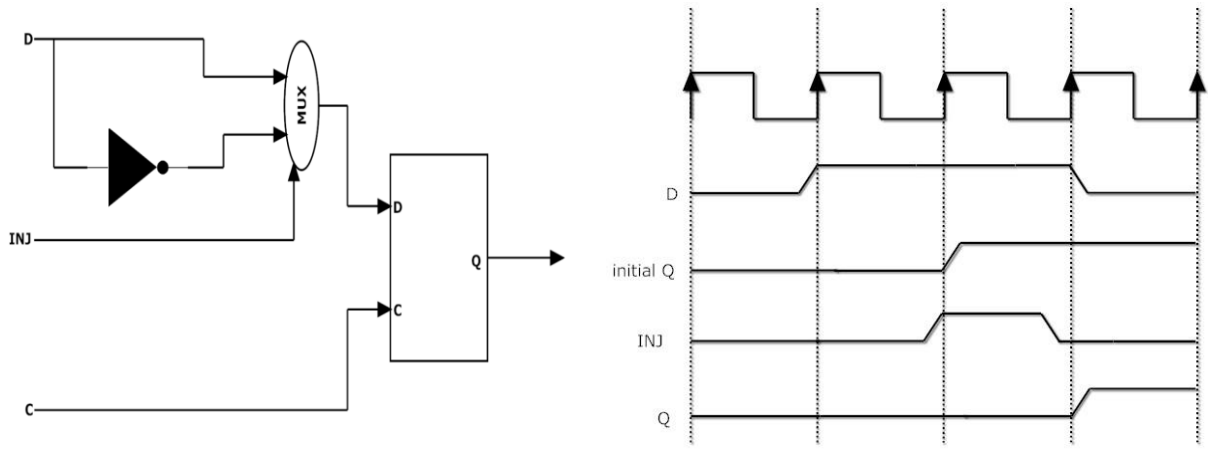


Figure 3.3 : La D flip-flop sans le signal d'activation modifiée

B. Les flips-flops avec un signal d'activation: Une écriture dans ce type de flip-flops nécessite l'activation du signal « *enable* ». La figure 3.4 montre une flip-flop avec un signal d'activation et la

modification nécessaire pour permettre l'injection de fautes. L'écriture dans la flip-flop sera faite si l'un des deux signaux « *CE* » (chip enable) ou « *INJ* » (injection) est actif. Si « *INJ* » et « *CE* » sont tous les deux actifs la donnée « *D* » inversée sera écrite dans la flip-flop. Si « *INJ* » est actif et « *CE* » est inactif on réécrit la sortie « *Q* » inversée. Dans le cas où le signal « *INJ* » est inactif la flip-flop garde son fonctionnement normal.

Il faut noter que toutes les autres flip-flops quelque soit leurs types (FD, FDC, FDE, etc.) sont dérivées de ces deux genres en ajoutant des signaux de contrôle tels "*reset*", "*set*" synchrones ou asynchrones etc. Toutes les flip-flops sont modifiées de la même manière.

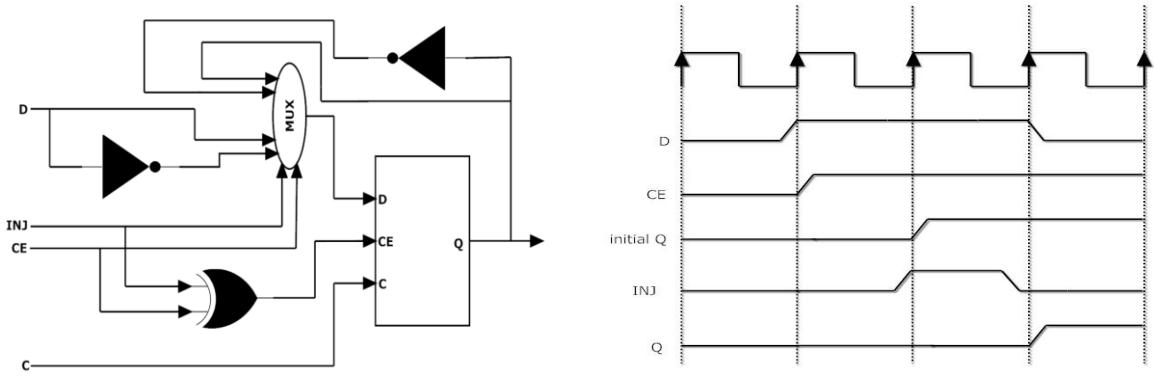


Figure 3.4 : La D flip-flop avec le signal d'activation modifiée

3.2.2 Modifications des BRAMs

Les Blocs RAMs (BRAMs) sont des composants importants dans les circuits complexes, surtout dans les processeurs. Ils sont généralement utilisés dans l'implémentation des mémoires caches et parfois dans les bancs de registres. Le nombre important de bits dans ces mémoires (8Kx32bits pour les mémoires caches dans un processeur LEON2), rendre impossible d'injecter des fautes bit à bit comme dans les cas des flip-flops.

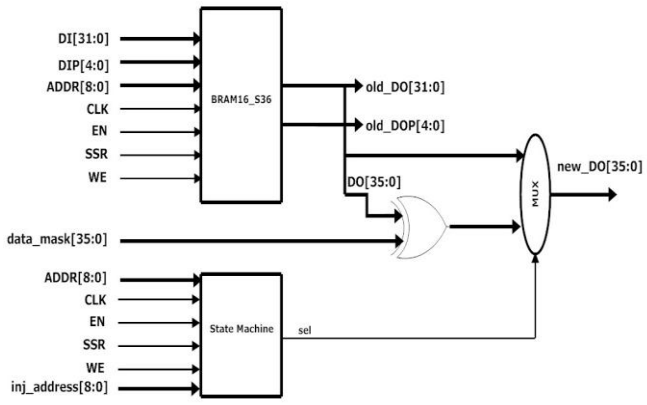


Figure 3.5 : Architecture modifiée du bloc-ram BRAM16_S36

La décomposition de la netlist quand elle contient des BRAMs est indispensable. L'injection des fautes dans les BRAMs sera faite en une seule campagne. Une machine à états, responsable de superviser l'état de l'adresse cible, est implémentée et exécutée en parallèle avec la BRAM. Cette machine à états permet d'émuler l'effet de la faute SEU dans ces BRAMs. A titre d'exemple, dans la figure 3.5 est illustrée l'architecture modifiée dans le cas d'une des BRAMs utilisée par Xilinx (la BRAM16_S36) dont les entrées/sorties sont: « *INJ* » signal d'injection, « *data_mask* » et « *inj_address* » qui représentent le bit et l'adresse à modifier, « *DO* » (32-bits data out), « *DOP* » (5-bits parity data out), « *ADDR* » (address), « *CLK* » (clock), « *DI* » (data in), « *DIP* » (parity data in), « *EN* » (enable), « *SSR* » (synchrounous set/reset), et « WE » (write enable).

La machine à états a six entrées et une seule sortie qui est le signal « *sel* » qui choisi entre la sortie de la BRAM et la donnée perturbée. La figure 3.6 montre les quatre états de cette machine. Si aucune écriture ou lecture de l'adresse cible n'est pas détectée, la machine à états reste dans son état initial, état S0. Si une lecture de l'adresse cible est détectée, la machine à états exécute une transition d'état vers S1 (*sel = 1*) pour un seul cycle permettant la perturbation de l'adresse cible, puis elle retourne à son état initial. Au cas d'une écriture au même instant d'une injection de la faute dans le même adresse cible, la machine perturbe la sortie de cette adresse (état S2) puisque dans cette BRAM la lecture est continuelle, et puis elle effectue une transition d'état vers un état *IDLE* (S3) afin que le signal d'injection soit désactivé.

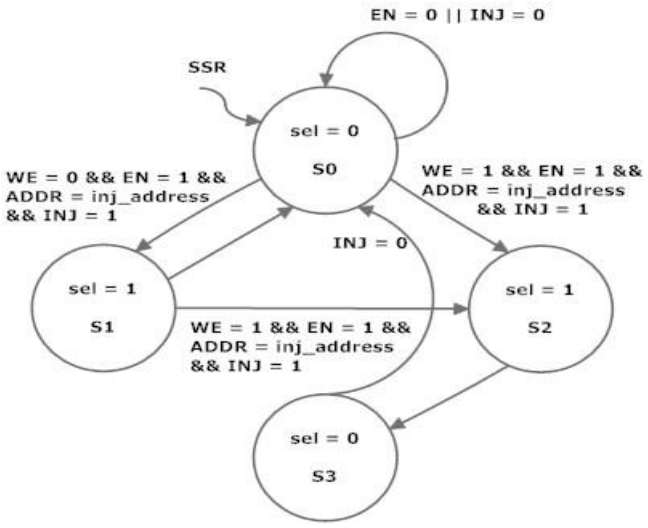


Figure 3.6 : Diagramme d'états de la machine à états pour la BRAM16_S36

Il faut noter que le BRAM16_S36 est seulement un exemple des BRAMs existant dans la bibliothèque built-in de Xilinx. Les modifications des autres mémoires BRAMs peuvent être réalisées par des mises-à-jour de cette machine à états.

3.2.3 Modifications des LUTs

Dans des FPGA SRAM, les LUTs sont utilisés pour l'implémentation des circuits combinatoires et ont différents nombres d'entrées. L'émulation de fautes de type SET et Stuck_at se base sur la modification de ces modules.

Un circuit combinatoire est composé de plusieurs portes logiques auxquelles on n'a pas accès au niveau de la netlist. Donc, l'injection de fautes pourra seulement être effectuée sur les sorties des LUTs, qui sont les nœuds les plus sensibles des circuits combinatoires. Ceci donne une estimation pire cas des effets des fautes SET et Stuck_at, estimation qui pourra cependant être utile pour la validation des architectures tolérantes aux fautes.

Pour les fautes de types SET, un inverseur et un multiplexeur sont ajoutés pour perturber le bit de la sortie des LUTs. Dans les cas de fautes stuck-at-0 et stuck-at-1, on perturbe la sortie pour qu'elle soit 0 ou 1 pour toute la durée de l'exécution. La figure 3.7 montre les différentes modifications appliquées à un LUT pour l'émulation des ces trois types d'erreurs.

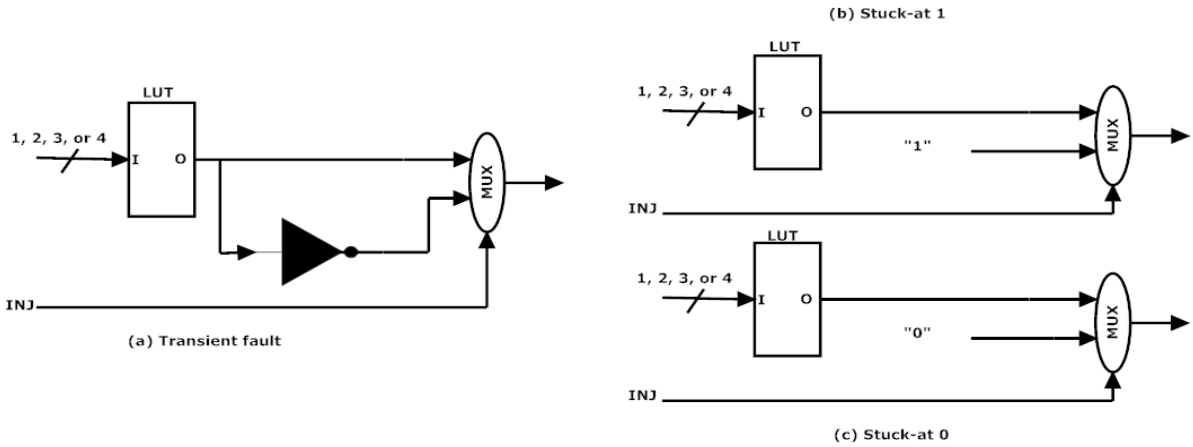


Figure 3.7 : Différentes modifications du LUT pour émuler de fautes (a) SETs, (b) Stuck-at-1 et (c) Stuck-at-0

3.3 Résultats expérimentaux

NETFI peut être utilisé pour l'étude de la sensibilité des circuits par rapport aux SEUs, ainsi que pour la validation des techniques de tolérance aux fautes. Dans le but de valider cette nouvelle méthode d'injection de fautes, des expériences utilisant NETFI ont été faites sur trois circuits différents.

Le LEON2 a été choisi, comme une première cible, pour illustrer l'application de la méthode NETFI et pour étudier la sensibilité, face aux SEU, de ce processeur lorsqu'il exécute un programme parmi les benchmarks utilisés dans le domaine d'injection de fautes : tri de 1024 données.

Les microcontrôleurs sont généralement des circuits qui ont une petite mémoire interne, et une petite largeur de bus de données. Notre deuxième cible a été le microcontrôleur 80C51 d'Intel qui est disponible en VHDL sur [Web opencores], et dont les résultats issus de tests sous radiations lorsque le circuit exécutait un autre programme benchmark, multiplication de matrices, ont été présentés dans [Rezgui 2001].

La dernière cible considérée pour ces expériences, a été un Réseau de Neurones Hopfield (RNH) dont le RTL fut conçu et implémenté sur FPGA dans le cadre de cette thèse. Deux versions de ce RNH ont été conçues d'une façon optimisée, l'une standard et l'autre incluant une technique de tolérance aux fautes originale. L'application exécutée par ces RNH était la reconnaissance des patterns alphabétiques. La méthode NETFI a été utilisée pour mettre en évidence la robustesse face aux fautes de types SEU, SET, et Stuck_at, du RNH incluant la tolérance aux fautes.

3.3.1 Le processeur LEON2

Le LEON2 est un modèle VHDL synthétisable VHDL d'un processeur 32-bit compatible avec l'architecture SPARC V8. Le modèle est hautement configurable, et particulièrement adapté pour la conception des systèmes-sur-puce (SOC). Le code source complet est disponible sous la licence GNU LGPL [web gaisler], permettant l'utilisation gratuite et illimitée dans les applications de recherche ainsi dans des applications commerciales.

Le processeur LEON2, dont l'architecture est donnée dans la figure 3.8, possède les caractéristiques suivantes [Web vlsicad]

- ALU compatible avec le jeu d'instruction SPARC V8
- 5 étages de pipeline
- Multiplicateur, diviseur et MAC matériels.
- Interface avec le FPU Meiko.
- Mémoire caches d'instructions de des données configurables et séparées (Hardvard l'architecture)
- Caches set-associative: 1 - 4 sets, 1 - 64 ko / set, remplacement aléatoire, LRR ou LRU
- Mode snooping pour la cache de donnée
- AMBA AHB-2.0 et APB bus on-chip
- Contrôleur de mémoire externe pour 8/16/32-bits PROM et SRAM
- Contrôleur pour 32-bits SDRAM PC133
- Périphériques on-chip tels que : UART, timers, contrôleurs d'interruption et 16-bit I/O ports
- Mode de débogage avancée disponible on-chip
- Mode power-down

La méthode NETFI, lance la synthèse du code VHDL du LEON2 pour obtenir la netlist qui sera modifiée pour permettre l'injection de fautes. Cette synthèse choisi le FPGA Virtex comme cible pour éviter l'utilisation des modules complexes comme le DSP48. L'analyse de la netlist montre qu'elle est constituée de 2613 flip-flops (pipeline et contrôle) et de 32 BRAM4_S8 dont 16 sont utilisés pour implémenter la mémoire cache de données et 16 pour la mémoire cache d'instruction. Le banc des registres, constitués de 135x32 bits, étant implémenté en utilisant les rams de type RAMX1D. Ces types de mémoires seront transformés en flip-flops pour injecter des fautes bit-à-bit. Leurs modifications seront montrées dans l'ANNEXE A.

Dans notre étude, les mémoires caches ne sont pas considérées comme cibles de l'injection de fautes, car les résultats issus des prédictions de taux d'erreurs avec la méthode NETFI seront confrontés, dans l'avenir proche, à ceux issus de tests sous radiations effectués sur un ASIC implémentant un processeur LEON2 dont les mémoires caches sont tolérantes aux fautes (ARTISAN RAM fourni par Atmel).

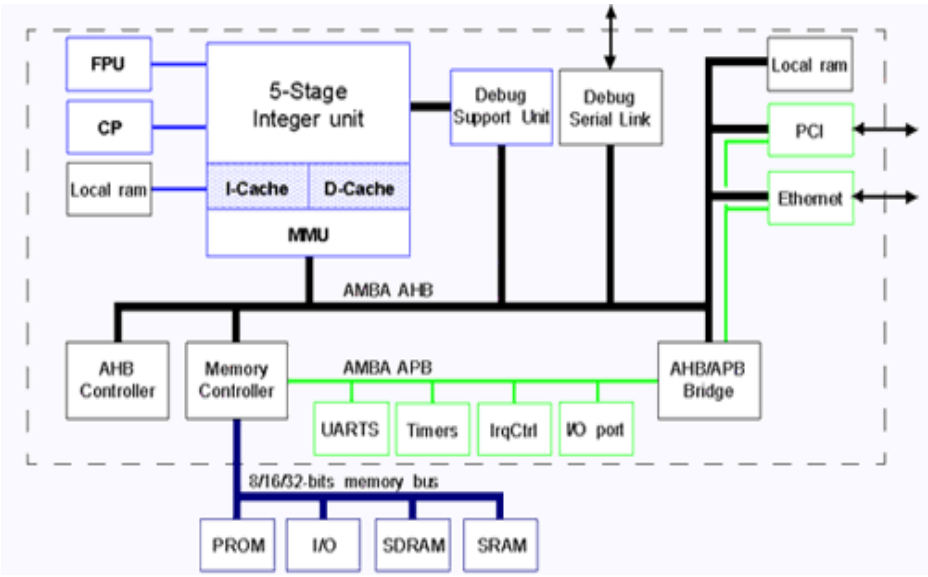


Figure 3.8: Architecture du processeur LEON2

L'analyse de la netlist du LEON2 générée par NETFI a permis d'identifier 6933 cellules mémoires dans ce design, cellules qui seront les cibles des campagnes d'injections de fautes avec la méthode étudiée. Ce grand nombre de cellules oblige à décomposer la netlist en deux sous-netlists, ceci afin de faire les modifications nécessaires pour pouvoir injecter des fautes. Deux campagnes d'injections de fautes, ciblant chacune une sous-netlist, doivent être réalisées. La première considère comme cible les flips-flops, tandis que la seconde cible les BRAMs (en excluant les BRAMs qui représentent les mémoires caches).

L'application exécutée par le LEON2 durant ces campagnes d'injection de fautes était un algorithme tri de 1024 données sauvegardées d'une manière décroissante au début de l'exécution de l'application. Le benchmark utilisé est donné par le code C dans la figure 3.9.

Deux campagnes d'injection de fautes ont été réalisées sur le processeur considéré, sans et avec l'activation des mémoires cache. Il est important de rappeler que les mémoires cache n'ont pas été ciblé par l'injection de fautes. Dans le tableau 3.1 est donnée la durée d'exécution obtenue par le *golden run*, exécution de l'application sans injection de fautes pour obtenir les résultats de référence. L'impact de l'activation des mémoires cache (de données et d'instructions) sur la durée totale de l'exécution du programme est clairement mis en évidence (106813736 cycle d'horloge seront gagné si les caches sont activées).

Tableau 3.1: Durée de l'exécution du programme *sorting* obtenue par le *golden run*

Option	Temps d'exécution
Cache désactivée	129577868 cycles d'horloge
Cache activée	22764132 cycles d'horloge

```
#define WORKLOAD  1024

void bubble_sort(int numbers[], int array_size);

int main(int argc, char **argv)
{
        int *result;
        int i, j;

       result = (int *)MEM_BASE;


       j = WORKLOAD-1;
       for(i = 0; i < WORKLOAD; i++) {
             result[i] = j;
             j--;
       }



       bubble_sort(result, WORKLOAD);


       return(0);

}
void bubble_sort(int numbers[], int array_size)
{
       int i, j;
       int temp=0;
       for (i = array_size - 1; i > 0; i--) {
             for(j = 1;j <= i; j++)
             {

                    if (numbers[j-1] > numbers[j]) {
                          temp = numbers[j-1];
                          asm("nop");
                          numbers[j-1] = numbers[j];
                          asm("nop");
                          numbers[j] = temp;
                          asm("nop");


                    }

             }
       }


}
```

Figure 3.9: Le code « C » du programme sorting

Les résultats issus des campagnes d'injection de fautes sur la zone sensible, considérée dans cette étude, sont donnés dans le tableau 3.2.

Tableau 3.2: Résultats des campagnes d'injection de fautes sur le processeur LEON2

Option	# des fautes injectées	# d'erreurs	# des timeouts
Cache desactivée	19474	630 (3.325%)	1832 (9.40%)
Cache activée	19867	526 (2.648%)	2125 (10.69%)

Le même nombre des SEU ont été injectés dans la zone sensible du processeur LEON2, durant les deux campagnes d'injections de fautes considérées. Le taux d'erreurs global (erreurs + timeouts) étant 13.34% dans le cas où la cache est activée, et 12.64% quand la cache est désactivée.

Cette expérience illustre l'utilisation de la méthode NETFI sur un processeur LEON2 sur lequel on prévoie de réaliser, dans la future proche, des campagnes de tests sous radiations pour valider la méthode sur des processeurs complexes. Dans le même but, un microcontrôleur 80C51, dont les résultats issus de tests sous radiation sont disponibles dans [Rezgui 2001], a été choisi comme une deuxième cible.

3.3.2 Le microcontrôleur Intel 80C51

L'estimation du taux d'erreurs d'un circuit intégré face aux effets des radiations est l'un des objectifs importants de la méthode NETFI. Le microcontrôleur 80C51 d'Intel, décrit en VHDL dans [Web opencores] a été choisi pour illustrer cet aspect. La disponibilité des résultats de tests sous radiations obtenus pour ce circuit dans [Rezgui 2001] permettra de confronter les prédictions du taux d'erreurs faites à partir des expériences d'injection par NETFI à ceux issus des tests sous faisceaux de particules et valider donc la méthodologie d'injection de fautes étudiée dans le cadre de cette thèse.

Le microprocesseur Intel 8051 correspond à la famille MCS51 [Souchard 1999] dont les principales caractéristiques sont:

- CPU de 8 bits optimisé pour le contrôle d'applications.
- Processeur booléen permettant le calcul sur un bit.
- 64K octets d'espace mémoire de programme.
- 64K octets d'espace mémoire des données.
- Mémoire programme intégrée d'une taille supérieure à 32KOctets.
- Mémoire interne RAM de 128 octets.
- Lignes d'entrées/sorties individuelles et bidirectionnelles.
- Communication bidirectionnelle UART.

Dans la figure 3.10 sont donnés les principaux blocs de l'architecture interne du 8051. Les zones sensibles aux SEUs de ce circuit sont: les 128 octets de la mémoire SRAM, les registres du « *Special funtion registers* » SFR, tous les registres du pipeline, le PC « *Program Counter* » et le

SP « *Stack pointer* ». NETFI peut injecter des fautes dans 100% de ces cellules dont le total fait 1633 bits.

Le programme choisi comme application pour l'injection de fautes a été une multiplication des matrices de dimensions 6x6, comme dans le PSOC présenté dans le chapitre 2, le choix de ce programme permet d'occuper environ 92% de la mémoire interne du microcontrôleur augmentant donc la probabilité d'observer des erreurs. Ce programme est le même que celui utilisé dans le cas des expériences de test en accélérateurs de particules présentés dans [Rezgui 2001].

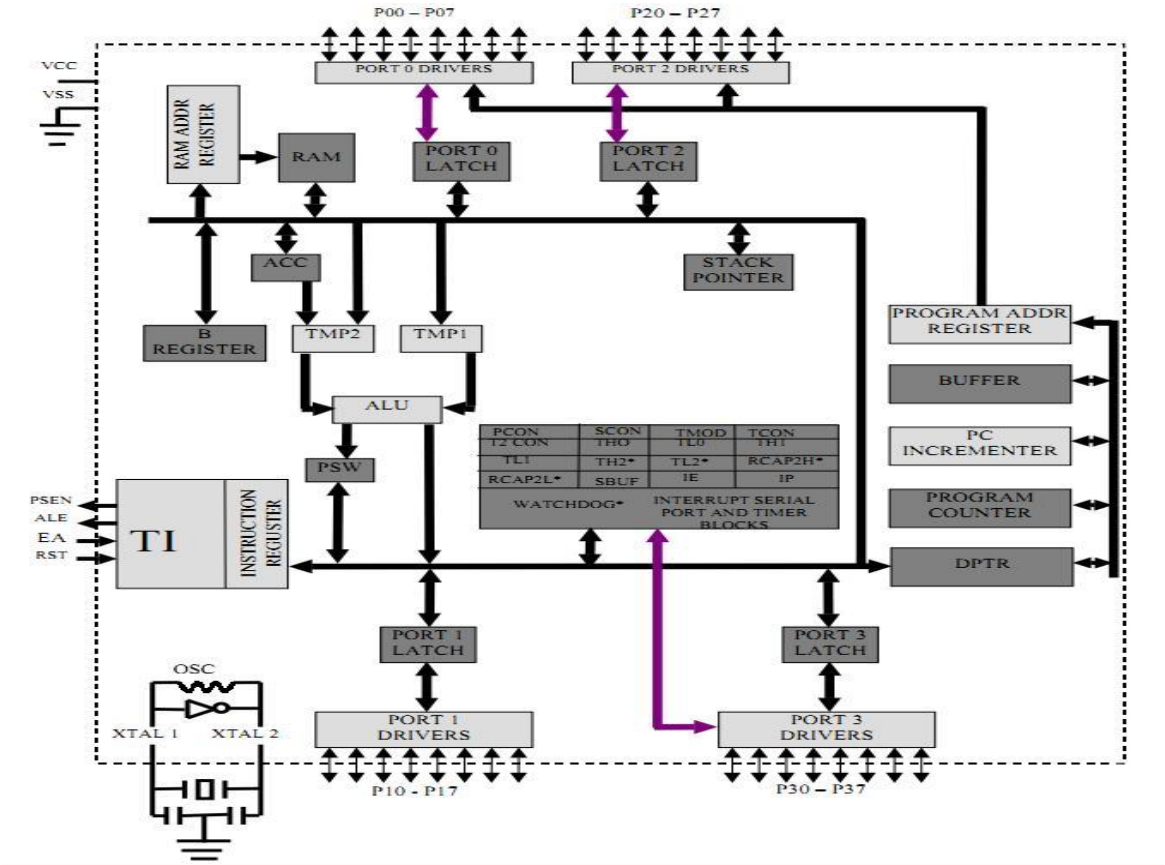


Figure 3.10: Architecture interne du 80C51

Dans le chapitre 2, il a été montré que la section efficace dynamique d'un circuit de type processeur exécutant un programme donné peut être prédite d'une manière précise par l'équation 3.1 :

$$\tau_{SEU} = \sigma_{SEU} * \tau_{NETFI} \quad \text{(Eq 3.1)}$$

où,

τ_{SEU} = section efficace dynamique prédite pour le programme considéré.

σ_{SEU} = section efficace statique des SEUs issue des tests sous radiations.

τ_{NETFI} = taux d'erreur globale (erreurs + timeouts) obtenus par injection des SEUs en utilisant la méthode NETFI.

Deux accélérateurs de particules fournissant plusieurs types d'ions à des LETs différents ont été utilisés dans [Rezgui 2001], le Tandem Van der Graff de l'IPN, à Orsay et le cyclotron de l'UCL à Louvain la Neuve en Belgique pour obtenir des mesures sur la section efficace statique et dynamique du 80C51. Le tableau 3.3 résume les résultats obtenus dans ces tests pour les ions utilisés. Le dépôt d'énergie, dans les circuits cibles, de ces particules est donné par leur LET qui dépend de l'ion et de l'angle d'incidence. Ce LET appelé le LET effectif (LET_{eff}) est calculé comme suit :

$$LET_{eff} = \frac{LET_0}{cos\theta} \qquad \text{(Eq 3.2)}$$

où :

θ : L'angle incident du faisceau d'ions,

LET_0 : Le LET nominal à l'angle 0,

LET_{eff} : Le LET effectif.

La section efficace statique a été déterminée par des campagnes de radiations lors duquel le processeur exécutait une application de type test statique (vérification continuelle du contenu de la mémoire interne et des registres) du microcontrôleur 80C51 exposé à des faisceaux de différentes particules d'ions lourds produits par les accélérateurs précédemment cités.

La prédiction de la section efficace dynamique basée sur la stratégie présentée dans l'équation 3.1 requière la prédiction du taux d'erreurs issu d'une campagne d'injection de fautes réalisée avec la méthode NETFI. Les résultats donnent un taux d'erreur de 47,09 % après 51907 fautes injectées.

La section efficace dynamique prédite calculée en utilisant l'équation 3.1, se servant de la section efficace statique mesurée, est donnée dans la dernière colonne du tableau 3.3.

Tableau 3.3: Prédiction NETFI Vs. Mesures des tests sous radiations

Ion	**LET [MeV/mg/cm^2]**	**Angle (degrés)**	**LET eff.**	**Section efficace statique [cm^2/composant]**	**Section efficace dynamique [cm^2/composant]**	
				σ_{SEU}	Mesurée	Predite $\tau_{SEU} = \sigma_{SEU} * \tau_{NETFI}$
N	2.97	0	2.97	$4.30\ 10^{-6}$	$2.00\ 10^{-6}$	$2.03\ 10^{-6}$
Ne	5.85	0	5.85	$3.33\ 10^{-4}$	$1.02\ 10^{-4}$	$1.58\ 10^{-4}$
Cl	12.7	0	12.7	$8.12\ 10^{-4}$	$3.96\ 10^{-4}$	$3.84\ 10^{-4}$
Ar	14.1	0	14.1	$9.31\ 10^{-4}$	$4.50\ 10^{-4}$	$4.40\ 10^{-4}$
Cl	12.7	48°	19.5	$1.29\ 10^{-3}$	$6.63\ 10^{-4}$	$6.10\ 10^{-4}$
Cl	12.7	60°	25.4	$1.62\ 10^{-3}$	$7.13\ 10^{-4}$	$7.68\ 10^{-4}$
Kr	34	0	34	$1.90\ 10^{-3}$	$9.12\ 10^{-4}$	$9.00\ 10^{-4}$
Br	40.7	0	40.7	$1.94\ 10^{-3}$	$8.85\ 10^{-4}$	$9.16\ 10^{-4}$

La bonne corrélation entre les prédictions et les mesures de la section efficace dynamique pour le DUT testé et le programme benchmark, figure 3.11, met en évidence l'efficacité de l'approche et de l'outil proposé (NETFI). En effet, pour toutes les particules utilisées la différence entre les sections efficaces dynamiques mesurées et prédites a été inférieure à 0.5x10-4. Il est important de noter que l'approche NETFI étant basé sur des émulations sur FPGA, elle permet d'explorer l'espace/temps de l'occurrence des SEUs d'une manière largement plus grande que les résultats de tests sous radiations. Pour obtenir les mêmes résultats sous faisceau il nous faudrait plusieurs dizaines d'heures, le coût [4] de ces campagnes de test étant élevé. Les tests sous radiation souvent ils sont arrêtés des que quelques dizaines d'erreurs ont été observés. L'exploration espace/temps de l'occurrence des SEU faite par l'injection de fautes est certainement plus complète que celle issus d'une campagne en accélérateur de particules et donc la prédiction est probablement plus proche du taux d'erreurs de l'application finale que la mesure issue de test sous radiations.

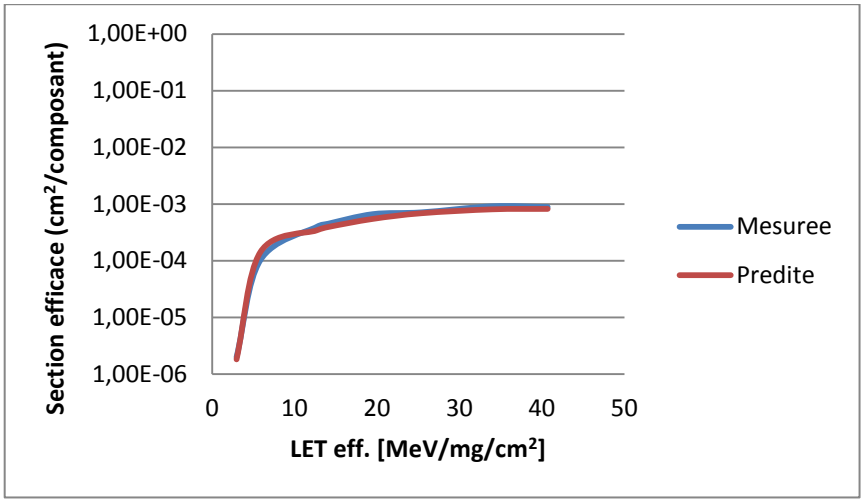


Figure 3.11 : Section efficace prédite et mesurée aux SEUs du 80C51

Une comparaison des prédictions de taux d'erreurs en utilisant deux stratégies d'injection de fautes différents : NETFI et CEU est donnée dans le tableau 3.4. Ceci montre que les résultats issus de ces deux méthodes sont en bonne corrélation. En effet, la différence du nombre des cellules sensibles accessibles par chaque méthode n'est pas très importante (1633 cellules sensibles dans le cas de NETFI et 1518 dans le cas de CEU environ 7%) ce qui explique la faible différence entre les valeurs obtenues. Cette différence peut être plus significative pour des processeurs plus complexes et sera illustré dans le chapitre suivant pour le cas d'un processeur LEON3.

[4] Environ 600 euros/heures incluant le temps lié à l'installation de la plateforme de test et la vérification de son fonctionnement avant de lancer les tests sous faisceau.

Tableau 3.4: Prédiction NETFI Vs. CEU

ion	LET	Section efficace mesurée	Section efficace prédite par CEU	Section efficace prédite par NETFI
			$\tau_{SEU} = \sigma_{SEU} * \tau_{CEU}$	$\tau_{SEU} = \sigma_{SEU} * \tau_{NETFI}$
N	2.97	2.00 10-6	$2.002\ 10^{-6}$	$2.03\ 10^{-6}$
Ne	5.85	1.02 10-4	$1.55\ 10^{-4}$	$1.58\ 10^{-4}$
Cl	12.7	3.96 10-4	$3.78\ 10^{-4}$	$3.84\ 10^{-4}$
Ar	14.1	4.50 10-4	$4.33\ 10^{-4}$	$4.40\ 10^{-4}$
Cl	12.7	6.63 10-4	$6.01\ 10^{-4}$	$6.10\ 10^{-4}$
Cl	12.7	7.13 10-4	$7.56\ 10^{-4}$	$7.68\ 10^{-4}$
Kr	34	9.12 10-4	$8.86\ 10^{-4}$	$9.00\ 10^{-4}$
Br	40.7	8.85 10-4	$9.01\ 10^{-4}$	$9.16\ 10^{-4}$

3.3.3 Réseau de neurones Hopfield (RNH)

Les Réseaux de Neurones Artificiels[5](RNA) sont considérés comme intrinsèquement tolérants aux fautes, car ils sont des modèles mathématiques inspirés par la structure de neurones biologiques [Rumelhart 1986]. Un RNA est constitué d'un ensemble de neurones interconnectés qui traitent des informations en utilisant les connexions entre eux.

Les RNAs sont devenus un sujet très dynamique et ils ont motivés de nombreuses recherches dans la dernière décennie [Saif 2006] [Saif 2007] [Stepanova 2007] [Leiner 2008]. Un des facteurs importants était le progrès de la technologie VLSI, ce qui rend plus facile l'implémentation de larges RNAs avec des moyens qui étaient indisponibles dans le passé, comme les FPGA. En effet, l'amélioration de la technologie VLSI rend réalisable, l'implémentation des systèmes parallèles avec des milliers de processeurs.

La capacité des FPGAs de nos jours permet la mise en œuvre de très grands réseaux de neurones, ayant de performances élevées grâce à l'architecture parallèle. Si les RNAs sont considérées pour des applications dont les fautes peuvent avoir des conséquences critiques (espace, biomédicale..), leur robustesse face à des fautes de types (SEU, SET et Stuck_at) doit être analysée.

Les RNAs peuvent être utilisés dans de nombreuses applications telles que la reconnaissance des formes, la reconstruction d'image à partir d'une image partielle, la suppression du bruit, et le récupération des informations. Des exemples représentatifs de l'utilisation de RNAs dans des projets spatiaux sont : classification des images de l'Antarctique prises par le satellite NOA11 [Kilpatric 1995], classification des nuages à partir d'images satellites prises par METEOSAT [Kwiatkowska 1995], détection des électrons de cavité de plasma à partir des mesures prises par le satellite FREJA [Waldemark 1995], la classification des *Whistlers* des protons et d'électrons à partir de mesures prises

[5] Artificial Neural Networks or ANN en anglais.

par le satellite AUREOL III [Miniere 1996], et le traitement des images du satellite SPOT1 [Cheynet 1999]. Les RNA peuvent être utilisés aussi dans le domaine de la robotique pour les installations nucléaires.

Dans notre recherche, on a implémenté un RNA de type Hopfield [Hopfield 1982] sur un FPGA qui a pour but de reconnaitre des patterns alphabétiques. L'implémentation a été réalisée par une architecture parallèle ayant une grande performance. L'algorithme utilisé requière seulement une seule opération de multiplication O(1), et (log N) opérations d'additions O(log N) tandis que la plupart des autres algorithmes requièrent O(N) multiplications et O(N) additions.

Cet algorithme a été modifié pour qu'il tolère les erreurs provenant des fautes dans ses unités arithmétiques. L'implémentation du RNA original ainsi que celle du RNA modifié sont détaillées dans les sous-sections suivantes. Dans le but de la validation de l'architecture tolérante aux fautes implémentée, des campagnes d'injection des différents types de fautes, SEU, SET et Stuck-at ont été réalisées par la méthode NETFI. La comparaison des résultats met en évidence l'efficacité de la technique de tolérance aux fautes proposée et ouvre une perspective potentielle à l'amélioration de ces techniques pour les utiliser dans des applications spatiales critiques.

3.3.3.1 Implémentation d'un RNH sur FPGA

Les RNHs sont des RNAs récurrents où les éléments de traitement sont les neurones. La sortie de chaque neurone est connectée à l'entrée de tous les autres neurones par l'intermédiaire des connexions, chacune ayant un poids synaptique. Ces poids « W » sont calculés en utilisant la règle de Hebb (eq 3.3) :

$$if\ i \neq j \quad W_{ij} = \sum_{p=0}^{r} x_i^p x_j^p$$
$$if\ i = j \quad W_{ij} = 0 \qquad \text{(Eq 3.3)}$$

où $0 < i,\ j < N + 1, \mathrm{X^p} = \{\mathrm{x_1^p}, \mathrm{x_2^p}, \ldots., \mathrm{x_N^p}\}$ et $\mathrm{x_i^p} \in \{-1,1\}$, p étant le nombre de bits dans un pattern de X et r étant le nombre total des patterns.

Pour fournir une conception efficace, avec peu de hardware mais rapide, on suppose que les patterns d'entrées a[i] sont des nombres binaires. Ensuite on applique l'équation 3.4 pour calculer un nouvel pattern d'entrée. Le système doit itérer jusqu'à ce qu'il soit stable, c.à.d. quand le nouvel pattern d'entrée $a_i[t]$ soit exactement égale à l'ancien $a_i[t-1]$.

$$a_i[t] = f\big(h_i(t)\big) = f\left(\sum_j W_{ij} a_j[t-1]\right) \qquad \text{(Eq 3.4)}$$

Cette équation propose les étapes de calcul suivantes qui sont réalisées pour $1 \leq j \leq N$:

Etape 1 : Distribuer $a_j[t-1]$ à tous les éléments de la colonne j de la matrice de poids W[t].

Etape 2 : Multiplier $a_j[t-1]$ par W[t] (figure 4.10).

Etape 3 : Additionner le résultat de la multiplication de l'étape 2 le long de chaque ligne de la matrice W pour calculer la somme $h_i[t]$ (figure 3.12).

Etape 4 : Appliquer la fonction d'activation $f(h_i[t])$ pour calculer $a_j[t]$.

On répète ces quatre étapes jusqu'on arrive à saturation. Ces étapes peuvent être exécutées en parallèle. La multiplication se fait en un seul cycle comme le montre la figure 3.12, puis l'addition prend (log n) cycles où n est le nombre des neurones du réseau Hopfield considéré. La figure 3.13 montre comment se fait l'addition dans une ligne de la matrice W.

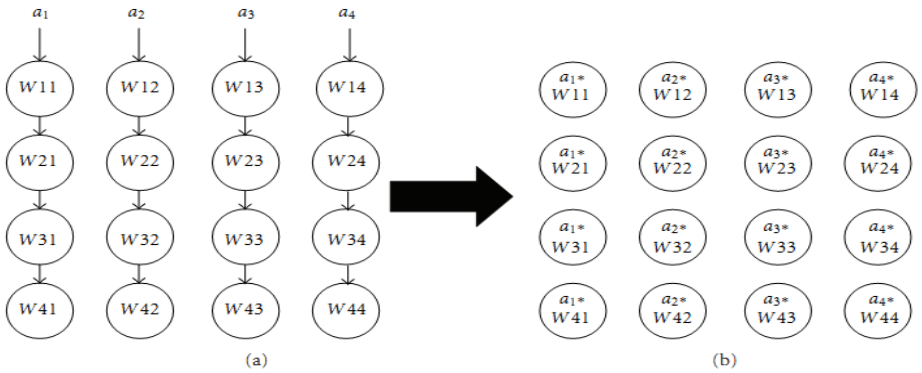


Figure 3.12: Multiplication parallèle pour un RNH de quatre nœuds

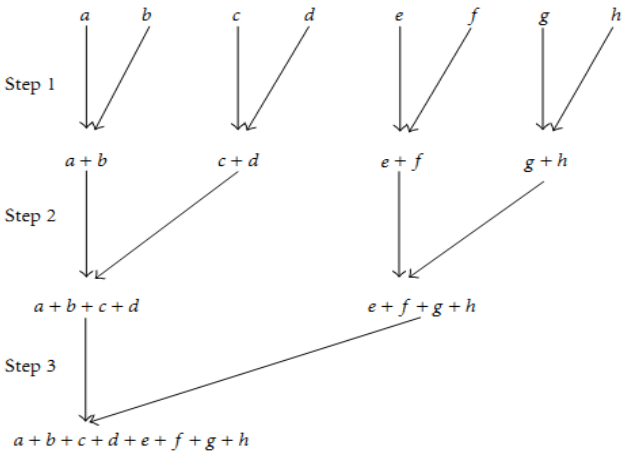


Figure 3.13: Addition de 8 cellules d'une ligne de la matrice de poids W

Dans les références [Ayoubi 2002] et [Ayoubi 2004] est donné une description détaillée de cet algorithme.

Pour l'implémentation de ce RNH, il faut considérer deux phases essentielles : la phase d'apprentissage et la phase de reconnaissance.

Durant la phase d'apprentissage, le RNH doit stocker toutes les combinaisons qui doivent être reconnues dans la phase de reconnaissance. Ce processus sera exécuté en série une fois pour chacun des patterns de l'ensemble considéré. Donc pour N patterns, N cycles d'horloge sont nécessaires pour terminer cette phase.

Un RNH de N nœuds nécessite N^2 unités d'apprentissage pour qu'il soit capable de calculer la matrice de poids W de dimension NxN en basant sur la règle de Hebb (eq 3.1).

Comme nos entrées sont des nombres binaires, elles doivent être implicitement transformées dans le design pour que le calcul soit effectué sur '-1' et '1', ceci pour être compatible avec les entrées calculées d'après la règle de Hebb qui sont -1 et 1 en hexadécimal 0xFFF et 0x001. Par hypothèse, on a fait le calcul des éléments de la matrice de poids à partir de la règle de Hebb sur 8-bits, mais pour éviter le débordement (overflow) on les a représentées par 12-bits.

La symétrie de la matrice de poids permettra de mettre en œuvre un système d'apprentissage qui exige seulement $((N^2/2) - N)$ unités d'apprentissage. La figure 3.14 montre l'architecture d'une unité d'apprentissage et sa distribution dans le réseau.

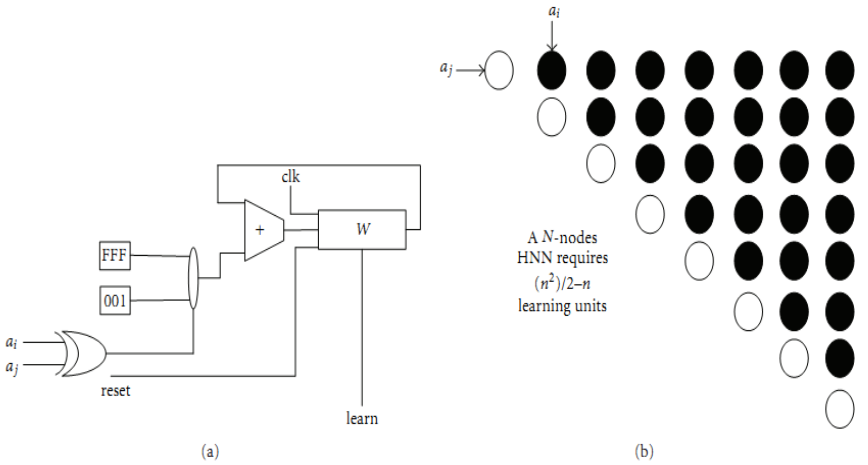


Figure 3.14: Architecture d'une unité d'apprentissage et sa distribution dans le RNH

La phase de reconnaissance est la seule phase considérée dans nos campagnes d'injection de fautes. Elle est constituée de deux types de nœud: le nœud série NS et le nœud maître NM. Le NS est utilisé pour effectuer l'étape de multiplication (a*W) puis une étape d'addition de deux nœuds adjacents (a1W1 + a2W2). NM est utilisé pour additionner les sorties de deux NS adjacents (a1W1 + a2W2 + a3W3+a4W4) ou pour additionner les sorties de deux NM adjacents (a1W1 + a2W2 + a3W3+ ... + a8W8).

- Le nœud série (NS): La multiplication du pattern d'entrée *a* par le poids *W*, peut être effectuée par l'intermédiaire d'un multiplexeur qui choisi soit le poids, si *a* = 1, soit son complément à 2 (2's complément), si *a* = -1 . Le résultat est sauvegardé dans un registre de 12-bit *parallel-in serial-out shift register* (PISO_SHR). La sortie série de ce registre sera additionnée à la multiplication du pattern adjacent avec le poids correspondant. À la fin, le résultat de cette addition est enregistré dans *un serial-in serial-out parallel-out shift register* (SISOPO_SHR) comme indiqué dans la figure 3.15.

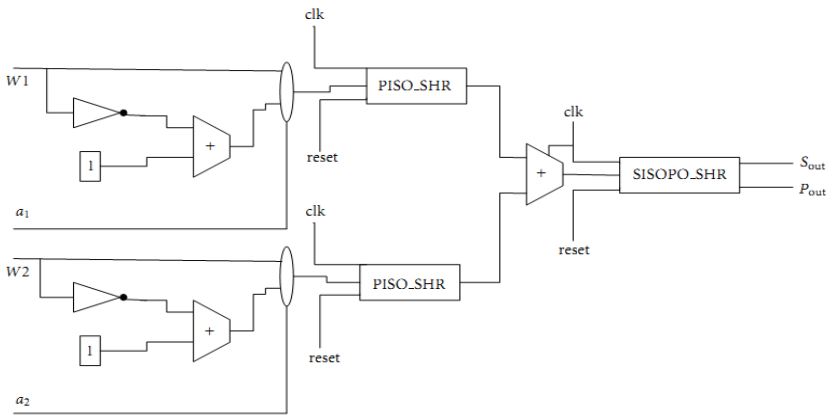


Figure 3.15: Architecture d'un nœud série (NS)

- Le nœud maître (NM) : Comme les RNH sont censés être mis en œuvre sur des larges réseaux, dans la première étape d'addition un NM devrait être utilisé pour chaque paire de NS adjacents, tandis que dans les autres étapes un NM sera utilisé pour chaque paire de NM adjacents. La figure 3.16 montre l'architecture du NM. la tache principale est l'addition de deux bits d'entrée et le stockage du résultat dans un SISOPO_SHR similaire à celui utilisé dans NS.

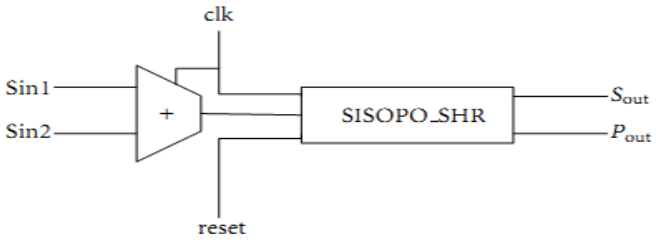


Figure 3.16: Architecture d'un nœud maître (NM)

- L'architecture d'une Ligne (Row Architecture) : Pour le calcul d'une ligne constituée de N nœuds, les NS sont toujours utilisés dans la première étape de calcul tandis que dans toutes les autres étapes on utilise les NM. Les figures 3.17 et 3.18 illustrent des lignes de quatre et huit nœuds, respectivement.

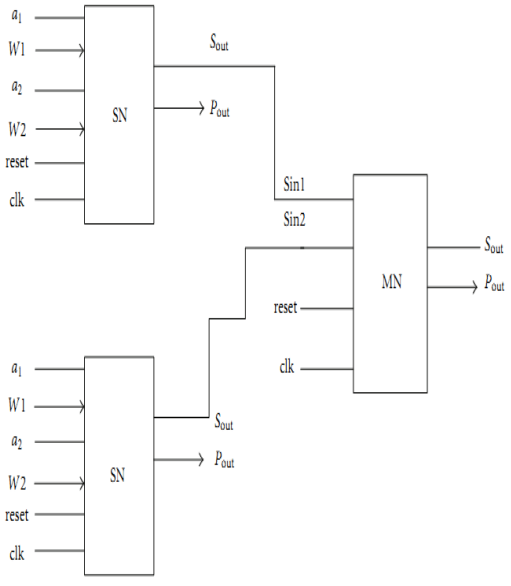


Figure 3.17: Architecture d'une ligne de 4 nœuds

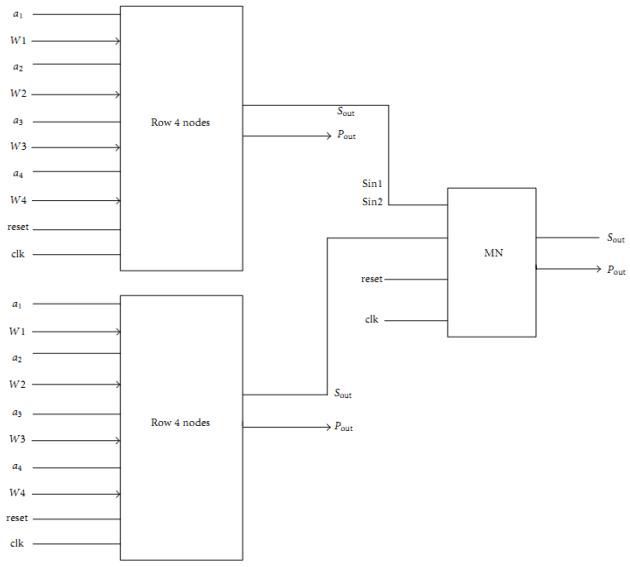


Figure 3.18: Architecture d'une ligne de 8 nœuds

- La dernière itération : Pour arrêter le processus de reconnaissance de patterns on doit attendre jusqu'à ce que le pattern d'entrée précédent soit exactement égal au nouvel pattern, ce qu'on appelle stabilisation. Le pattern d'entrée initial n'est utilisé que dans la première itération mais le pattern

recalculé sera utilisé dans les itérations qui suivent. Ce travail sera effectué par une machine à états qui lit la sortie du réseau, la vérifie, calcule le nouveau pattern d'entrée, et l'envoie au réseau jusqu'à ce que la condition d'arrêt est satisfaite. La figure 3.19 présente une maille (*mesh*) de 8×8 connectée à la machine à 'état (SM).

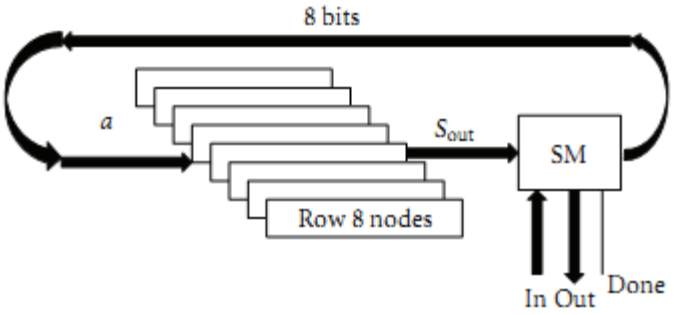


Figure 3.19: Architecture d'une maille de 8x8

3.3.3.2 Implémentation d'un RNH tolérant aux fautes sur FPGA

Le RNH décrit dans la section précédente a été modifié pour qu'il soit capable de tolérer les erreurs dues aux effets des radiations telles les SEUs et les SETs. Comme la multiplication et l'addition sont les opérateurs principaux pour le RNH, la tolérance aux fautes étudiée est basée sur la stratégie suivante :

- La répétition de la multiplication pour détecter et corriger les erreurs dans les multiplicateurs.
- La réalisation du processus d'addition « n » fois (n étant le nombre des cellules dans une ligne).

En d'autres termes, la redondance temporelle est utilisée pour les multiplications, tandis que la redondance spatiale (d'ordre "n") est adoptée pour les additions. Ce travail considère les additions et les multiplications en tant que cibles de durcissement. La figure 3.20 illustre la mise en œuvre proposée pour les multiplicateurs et les additionneurs robustes par rapport à différents types des fautes pour le RNH tolérant aux fautes (RNH-TF).

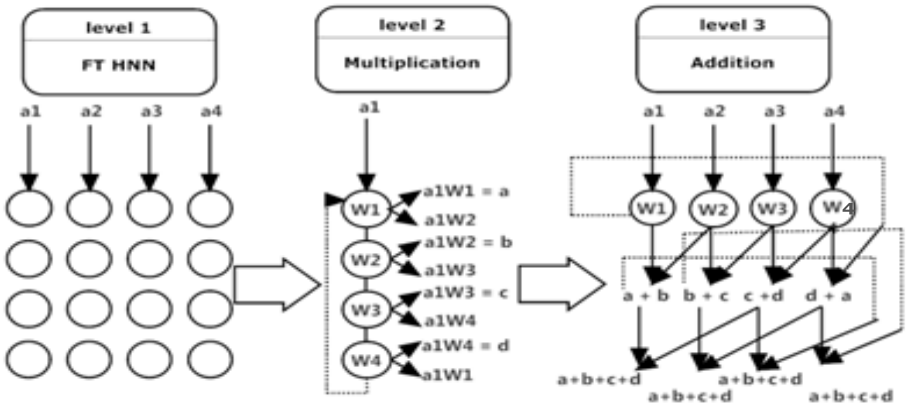


Figure 3.20: Le processus de multiplication et d'addition pour 4 nœuds adjacents

Pour l'implémentation de ce RNH, il faut considérer les deux phases essentielles : la phase d'apprentissage et la phase de reconnaissance.

Concernant la phase d'apprentissage, le RNH doit stocker toutes les combinaisons qui doivent être reconnus dans la phase suivante. Ce processus sera exécuté en série pour l'ensemble de patterns et pour une seule fois. La phase d'apprentissage décrite dans la section précédente, est implémentée une seule fois et conduit au calcul de la matrice de poids. Dans ce travail, nous considérons que la phase d'apprentissage n'est pas une cible d'injection de fautes, donc dans ce qui suit la phase de reconnaissance est la seule cible considérée.

Deux types de cellules sont utilisés pour la mise en œuvre de la phase de reconnaissance pour l'architecture RNH-TF proposée le nœud série tolérant aux fautes et le nœud maitre:

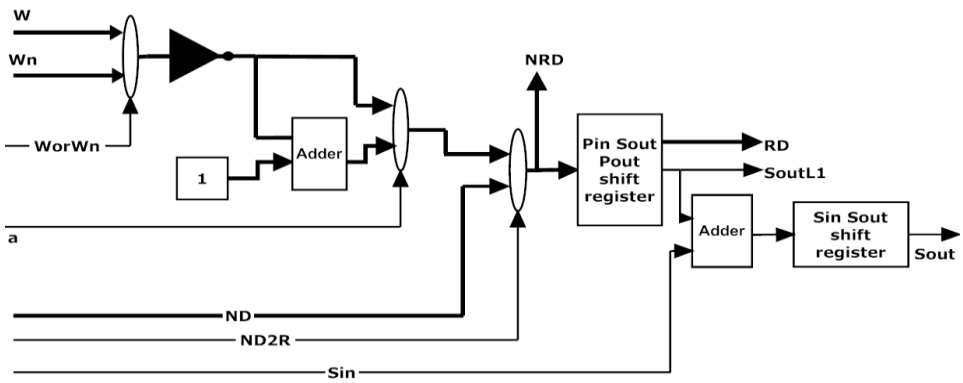


Figure 3.21: Architecture du nœud série tolérant aux fautes NSTF

- Le nœud série tolérant aux fautes (NSTF) est responsable au premier abord du processus de multiplication. Deux multiplications dont les résultats seront comparés pour l'identification des erreurs transitoires éventuelles seront réalisées successivement. Un signal « *select line* » est ajouté au multiplicateur pour permettre cette redondance temporelle. Des signaux et des multiplexeurs sont aussi

ajoutés: « *WorWn* », permet de choisir soit le poids local de 12-bits, W, soit le poids voisin de 12-bits, « *Wn* », provenant de la cellule adjacente dans la même colonne. Un multiplexeur responsable de fournir le résultat du processus de multiplication basé sur le pattern d'entrée « *a* » est implémenté. La figure 3.21 montre le diagramme de NSTF proposé dans cette conception.

Afin de tolérer les fautes dues à des erreurs de multiplication, le NSTF prend une entrée de 12-bits appelé « *neighbor data* », *ND*, où le *ND* est connecté à une donnée non enregistrée, appelée « *not registered data* » *NRD*, du NSTF voisin dans la même colonne. « *Neighbor_Data_To_Register* » ou *ND2R*, choisi le résultat qui doit être enregistré dans le 12-bit PISOPO_SHR.

Une machine à états chargée de générer le *WorWn* et *ND2R* a été mise en œuvre pour chaque colonne. Elle prend comme entrée tous les *RD* et *NRD* des cellules qui sont voisines dans une colonne. La deuxième tâche de la NSTF est l'addition des données locales multipliées avec d'autres données provenant d'un NTSF voisin (a1W1 + a2W2). Pour ce but, un additionneur est mis en œuvre pour réaliser cette addition entre la sortie du PISOPO_SHR et une entrée série « *serial input* », *Sin*, à NSTF. *Sin* est aussi la sortie *SoutL1* du PISOPO provenant de la cellule suivante. Les résultats finaux sont stockés dans un serial-in serial-out SISO_SHR.

- Le nœud maître (NM) : Le NM utilisé dans cette conception possède la même architecture que le NM utilisé dans la section précédente. La seule différence est que dans l'implémentation tolérante aux fautes, une ligne de « n » nœuds a besoin de $n*\log_2(n)$ des NMs.
- L'architecture d'une ligne (One Row Architecture) : Pour la mise en œuvre d'une ligne de n-nœuds, NSTFs sont toujours utilisés dans la première étape de calcul, tandis que dans toutes les autres étapes on utilise les NMs. Une ligne de "n" cellules nécessite (n^2-n) NMs. La figure 3.22 montre la mise en œuvre d'une ligne de 8 nœuds.
- La dernière itération : La même machine à états (figure 3.19) est utilisée dans ce design pour détecter la condition de stabilité du RNH.

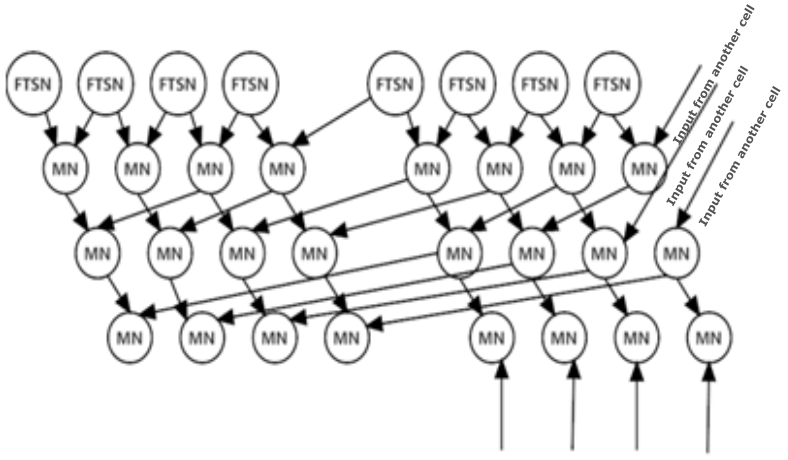


Figure 3.22: Architecture d'une ligne de 8 nœuds

3.3.3.3 Résultats expérimentaux des injections de fautes sur les RNHs

La méthode d'injection de fautes NETFI a été utilisée pour comparer la sensibilité aux fautes des deux versions du RNH. L'application benchmark utilisée est la reconnaissance de patterns alphabétiques, où chaque pattern est représenté par 32-bits, figure 3.23. Les deux implémentations montrent une bonne performance dans le processus de reconnaissance : 17 cycles sont nécessaires pour compléter une itération de reconnaissance dans le RNH original, alors que le modèle tolérant aux fautes nécessite 21 cycles.

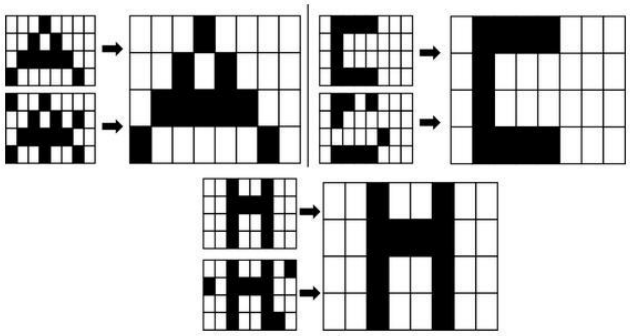


Figure 3.23: Entrées et sorties des RNHs

Les deux modèles ont été implémentés sur le C*hipset FPGA* XC4VLX40 VirtexIV de la plateforme ASTERICS pour effectuer l'injection de fautes. Une comparaison en termes de ressources matérielles utilisées est donnée dans le tableau 3.5 montrant que la version tolérante aux fautes requière environ 2,2 fois plus de slices, 2,4 fois plus de flip-flops et 2,2 fois plus de LUT que la version non tolérante aux fautes. Ceci affecte à son tour la fréquence maximale du système cible qui est dans le cas de la version originale, non tolérante aux fautes, 157 MHz et pour la version modifiée, tolérante aux fautes, 104 MHz.

Tableau 3.5: Comparaison matérielle entre les deux types de RNHs

Type	RNH Tolerant Aux Fautes	RNH
Maximum frequency	104.811 MHz	157.604 MHz
# Slices	6210 (33%)	2803 (15%)
# Slice flip flop	8325 (22%)	3449 (9%)
# 4 input LUTs	10593 (28%)	4800 (13%)

Plusieurs campagnes d'injection de fautes, au cours desquelles NETFI injecte un très grand nombre de fautes dans les deux designs, ont été réalisées. Il est important de noter que dans chaque exécution une seule erreur a été injectée dans la zone sensible. Les conséquences de fautes injectées sont classées comme suit:

- Silencieuse (silent fault): si la faute injectée n'a pas d'effet.
- Erreur: si la sortie du RNH n'est pas celle attendue.
- Délai d'attente (ou Timeout): si le RNH n'arrive pas à stabilisation.
- Converge: si le RNH fournit des résultats corrects après le temps limite prévu en cours d'exécution.

La première campagne d'injection de fautes a été réalisée sur le RNH original. Différents types d'erreurs ont été injectées dans le design lorsqu'il est configuré, ceci pour reconnaître un des patterns perturbé "A", "C" et "H" parmi plusieurs alphabets utilisé dans la phase d'apprentissage. La durée du processus de reconnaissance obtenue lors de la *golden run* est de 34 cycles c.à.d. deux itérations. Le tableau 3.6 présente les résultats de quatre campagnes d'injection de fautes, où les fautes de types SEUs, SETs, et Stuck-at sont considérés.

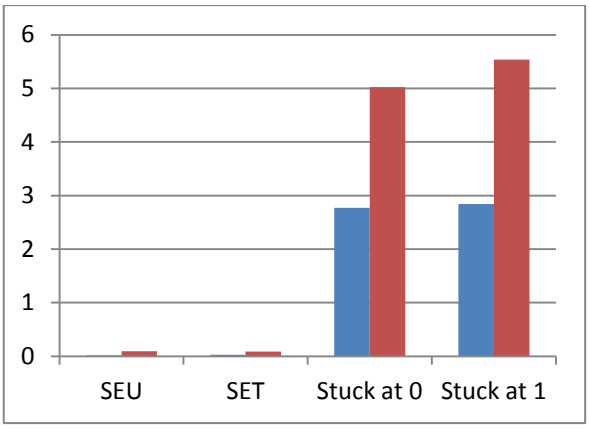
Tableau 3.6: Résultats des injections de fautes sur le RNH initial

Type of faults	# Inj. Faults	Results Errors	Timeouts	Converges
SEU	81831	79(0.096%)	1658(2.026%)	10865(13.277%)
SET	125676	110(0.087%)	2623(2.087%)	15195(12.091%)
Stuck at 0	190509	9566(5.021%)	8417(4.418%)	19393(10.179%)
Stuck at 1	98316	5444(5.537%)	3076(3.128%)	7659(7.790%)

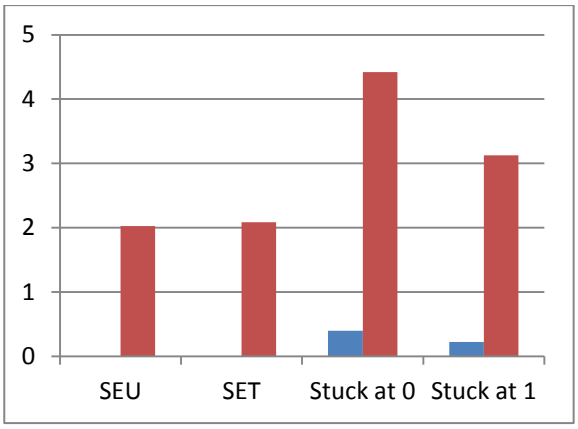
Les mêmes tests ont été effectués sur la version tolérante aux fautes. Le temps d'exécution obtenu lors de la *golden run* pour les mêmes patterns d'entrées est deux itérations de 42 cycles. Le tableau 3.7 montre les résultats de quatre campagnes d'injection effectuées sur le RNH tolérant aux fautes.

Tableau 3.7: Résultats des injections de fautes sur le RNH tolérant aux fautes

Type	# Inj. Faults	Results Errors	Timeouts	Converges
SEU	137801	20(0.014%)	6(0.004%)	4789(3.475%)
SET	149198	44(0.029%)	4(0.002%)	4193(2.810%)
Stuck at 0	140516	3897(2.773%)	557(0.396%)	1667(1.186%)
Stuck at 1	151803	4318(2.844%)	337(0.222%)	2486(1.637%)



Pourcentage des taux d'erreurs, (bleu) version tolérante aux fautes, (rouge) version originale



Pourcentage des taux des timeouts, (bleu) version tolérante aux fautes, (rouge) version originale

Figure 3.24: Comparaison des taux d'erreurs et des timeouts pour les deux RNH

La figure 3.24 montre clairement la comparaison entre les deux versions du RNH étudiée. Dans le cas d'injection de fautes SEUs et SETs, le taux d'erreurs global (erreurs + timeout) a diminué de 2,122% à 0,018% (environ 117 fois) pour les SEUs, et de 2,174% à 0,032% (environ 67 fois) pour les SETs. Cela montre l'efficacité de la conception durcie étudiée. Pour les fautes stuck-at, le gain était 6,269% (environ 3 fois) pour les stuck-at 0, et 5,599% (environ 3 fois) pour les stuck-at 1. Les convergences subissent une diminution due au fait que les calculs sont le plus souvent corrects dans le cas du design tolérant aux fautes. L'analyse approfondie des résultats sur la zone sensible du RNH tolérant aux fautes montre que toutes les erreurs se produisent lorsque la faute injectée cible la région où la machine à états global du système est implémentée.

Ceci prouve que les RNAs qui sont intrinsèquement robustes, doivent toutefois être durcis s'ils sont considérés pour des applications critiques. Les résultats obtenus mettent en évidence la fiabilité du circuit RNA tolérant aux fautes implémentés, et ouvre une perspective pour son application dans des réseaux plus larges et plus représentatifs des applications critiques.

3.4 Conclusions

Dans ce chapitre a été présenté une nouvelle méthode, appelée NETFI, pour l'émulation de fautes (SEU et SET) au niveau netlist. Cette méthode a été appliquée sur plusieurs circuits cibles. Un premier exemple pour illustrer l'application sur un processeur complexe, le LEON2, a été présenté. Une campagne d'injection de fautes de type SEU a été réalisée lorsque le processeur étudié exécute un programme de tri de 1024 données.

NETFI a été aussi utilisé pour estimer le taux d'erreurs provoquées par des SEU, d'un microcontrôleur 80C51 exécutant un programme de multiplication de matrices. Une bonne corrélation entre les résultats issus de cette prédiction et ceux issus de tests sous radiations, réalisées dans le cadre d'une thèse précédente, a été mise en évidence. Les taux d'erreurs issus des prédictions avec les méthodes NETFI et CEU ont aussi été comparés. Les bits accessibles par la méthode NETFI étaient 115 bits de plus que ceux accessibles par la méthode CEU. Cette faible différence justifie la bonne corrélation entre les résultats prédits par la méthode CEU et ceux prédits par NETFI.

Finalement, dans le but de valider des architectures tolérantes aux fautes, une dernière cible de NETFI a été un Réseau de Neurones de Hopfield, dont deux versions (une standard et l'autre incluant des techniques de tolérance aux fautes de type redondance matérielle et temporelle) implémenté dans le cadre de cette thèse sur un FPGA. Plusieurs campagnes d'injection de fautes (SEU, SET et Stuck_at) ont été réalisées sur les deux versions de ce réseau de neurones. Les taux d'erreurs globaux (erreurs + timeouts) issus de ces campagnes de tests mettent en évidence la robustesse de la version tolérante aux fautes implémentée.

Chapitre 4. Evaluation de l’efficacité d’un algorithme tolérant aux fautes : résultats issus des méthodes NETFI et CEU

Ce chapitre a pour but d'étudier la robustesse face aux SEU d'un algorithme, supposée tolérant aux fautes, dit *self-convergent*. Les méthodes CEU et NETFI seront utilisées pour injecter des fautes de type SEU dans les zones sensible d'un processeur LEON3 exécutant cet algorithme. Une analyse approfondie ayant pour but l'identification des variables les plus sensibles de l'algorithme considéré sera présentée. A la fin de ce chapitre, une comparaison entre les résultats issus des deux méthodes, NETFI et CEU, sera montrée.

4.1 L'algorithme self-convergent

La self-convergence est une propriété d'un système distribué, qui lui permet, quand il sera perturbé ou mal initialisé, de récupérer un fonctionnement correct en un nombre d'étapes de calcul fini. Ce concept est introduit par DIJKSTRA en 1974 [Dijikstra 1974] [Doley 2000].

Cette propriété permet à un système distribué de tolérer les défaillances, en particulier celles qui ne modifient pas le code exécuté. Après une perturbation, le système atteint une configuration arbitraire (les variables du programme peuvent avoir une valeur quelconque par exemple), mais il peut récupérer un comportement correct dans un temps fini si aucune nouvelle perturbation ne se produit.

L'algorithme contient deux configurations: légitimes et illégitimes (voir figure 4.1). La self-convergence est généralement considérée comme la combinaison de deux propriétés:

- Fermeture: à partir d'une configuration légitime, le système reste dans cette configuration pour toute la durée de l'exécution.
- Convergence: en partant d'une configuration arbitraire, le système atteint par la suite une configuration légitime.

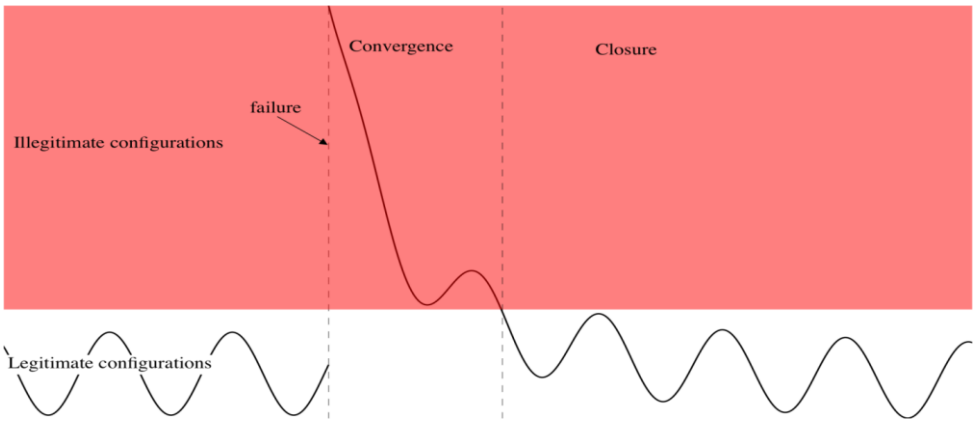


Figure 4.1: Comportement d'un système self-convergent

L'algorithme self-convergence utilisé dans le cadre de cette thèse est consacré au calcul de plus court chemin d'un graphe. Théoriquement, quelque soit l'erreur pouvant se produire au cours du calcul, le résultat de l'exécution sera toujours correct. C'est un problème classique d'informatique distribuée [Doley 1995] [Huang 2005].

Le code d'un algorithme self-convergent est donné dans la figure 4.2. L'entrée de cet algorithme est une matrice de dimensions NxN avec des valeurs prédéfinies, tandis que la sortie est une matrice dont la dimension est de NX1. Les variables booléennes b et c sont utilisées pour détecter la convergence de l'algorithme vers le résultat final. Les entiers i et j sont des indices de boucle requis pour effectuer les calculs matriciels.

```
b=c=1
T= NxN matrix
D= Nx1 matrix
while(b||c){
c=b;
b=0;
D[0]=0;
for(i=1; i<N; i++){
m = INFINIE;
for(j = 0; j<N; j++){
if(m>=D[j]+T[N*i+j])
m=D[j]+T[N*i+j];
}
if(D[i]!=m)
b=1;
D[i]=m;
}
}
```

Figure 4.2: Le code "C" de l'algorithme self-convergent

La matrice T représente le graphe dans lequel deux nœuds i et j sont reliés par une connexion de longueur T_{ij}. La distance entre un nœud i et le nœud de référence (nœud 0) est la seule quantité qui satisfait l'équation 4.1:

$$D_0 = 0;\ D_i = \min\{T_{ij} + D_j\} \qquad \text{Eq 4.1}$$

Le programme *self-convergent* définit D_i à $min\{T_{ij} + D_j\}$ pour tout i jusqu'à ce qu'il converge vers une solution, c.à.d. la seul matrice D qui satisfait l'équation 4.1. Cette solution est obtenue lorsque D ne sera pas modifié au cours des itérations. Dans ce cas, b sera égal à *"faux"* ou "0", et à l'itération suivante, c sera modifié à *"faux"* ou "0", conduisant à la fin du programme.

En principe, les seules erreurs qui peuvent conduire à des résultats incorrectes sont:

- Des erreurs qui provoquent la fin de la boucle avant le temps nécessaire pour terminer l'exécution de l'application; par exemple, quand les variables b et c sont égales à 0

conduisant *while(b||c) = 0*, quand l'ancienne valeur de *D* (valeurs calculées sur la boucle *while* précédente) est exactement la même que la nouvelle *D,* et quand *i* et/ou *j* ont des valeurs hors de la portée de leurs boucles;

- Les erreurs affectant les prochaines valeurs de *D*, qui se produisent après la fin de la boucle *while*.

Les deux booléens *b* et *c* ont été utilisés pour s'assurer qu'une seule erreur, sauf les erreurs qui touchent le compteur du programme, ne peut pas, en principe, conduire à un résultat erroné.

Les erreurs dans le compteur du programme peuvent conduire à une erreur si le compteur du programme saute indûment en dehors de la boucle, ou s'il saute vers une instruction et exécute un mauvais *opcode*.

L'algorithme self-convergence peut être utilisé dans les domaines de communications, surtout dans les processeurs à plusieurs cœurs pour garantir le transfert correct d'un message d'un cœur à un autre.

4.2 Le circuit cible: le LEON3

Le LEON3 [Web gaisler] est un modèle VHDL synthétisable d'un processeur 32-bit compatible avec l'architecture SPARC V8. Le modèle est hautement configurable, et particulièrement adapté aux systèmes sur puce (SOC). Le processeur LEON3 a les caractéristiques suivantes:

- Jeu d'instructions SPARC V8 avec l'extension V8e.
- Pipeline de 7-étages.
- Contient l'implémentation des unités multiplieur, diviseur, et MAC (Media Access Control).
- FPU (Floating Point Unit) IEEE-754 entièrement en pipeline et avec haute performance.
- Cache des instructions et cache des données séparées (architecture Harvard).
- Les caches sont configurables (1-4 ways, 1-256 kbytes/way set-associative, remplacement aléatoire, ou LRU (Least Recently Used)).
- Interface de bus AMBA AHB 2.0.
- Prise en charge avancée de débogage sur puce.
- Support multiprocesseurs symétrique (SMP).
- L'horloge peut aller jusqu'à 125 MHz en FPGA et 400 MHz sur 0,13µm technologies ASIC.
- Version tolérante aux fautes, et version SEU-proof disponibles pour des applications spatiales.

Le processeur LEON3 est entièrement paramétrisable grâce à l'utilisation des génériques VHDL. Ainsi, il est possible d'instancier plusieurs cœurs de processeur dans le même design avec des configurations différentes. Le LEON3 peut être configuré en utilisant un outil graphique construit sur *tkconfig* du noyau linux. Cela permet de définir rapidement une configuration adaptée sur mesure. Cet outil permet encore de configurer d'autres périphériques sur puce tels que les contrôleurs de mémoire et les interfaces réseau. La figure 4.3 montre l'architecture d'un LEON3 configurable, qui est configuré pour utiliser le JTAG debug link, le contrôleur AHB, le contrôleur mémoire, le bridge AHB/APB, l'UART, les TIMERS, les interruptions, et les ports I/O.

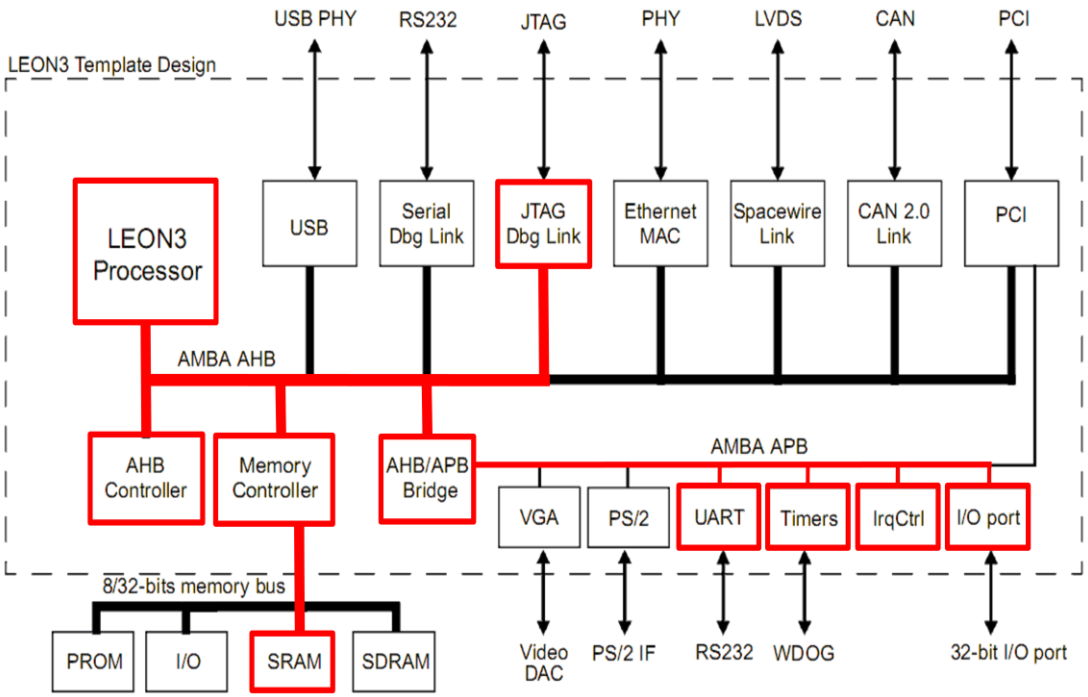


Figure 4.3 : Configuration d'un LEON3

4.2.1 Caractéristique de la fenêtre des registres

Le processeur LEON3 n'a pas un registre pointeur de pile unique comme les processeurs typiques. Il est organisé autour d'un système de 8 "fenêtres", chacun contient 32-registres de 32-bit distribués de la manière suivante:

- 8 registres locaux l0-l7
- 8 registres entrées (input) i0-i7
- 8 registres sorties (output) o0-o7
- 8 registres globaux (g0-g7) qui sont communs entre tous les fenêtres.

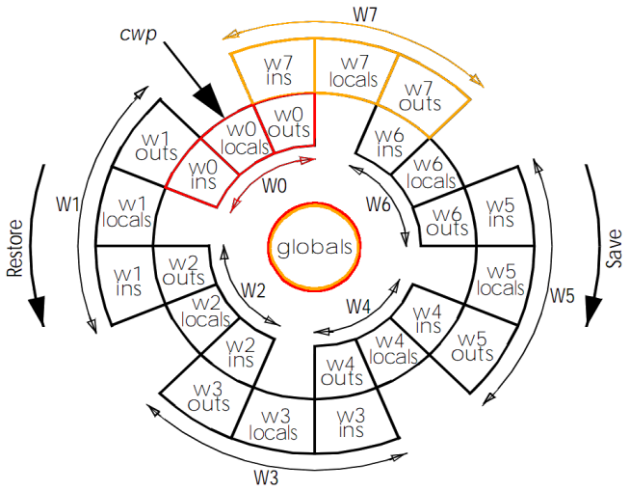


Figure 4.4: Fenêtres des registres du processeur LEON3

Le registre CWP (Current Window Pointer) indique quelle est la fenêtre active parmi les 8 fenêtres du processeur. La figure 4.4 montre les fenêtres des registres du LEON3. Parmi les jeux d'instructions, seulement deux instructions permettent le passage d'une fenêtre à une autre, le *restore* et le *save*. En plus, une interruption ou un appel d'une fonction provoquent un changement de fenêtre.

4.2.2 Zones sensibles aux SEU du LEON3

Dans le but d'étudier la robustesse de l'algorithme self-convergent, plusieurs campagnes d'injections de fautes, avec les méthodes CEU et NETFI, seront présentées dans ce chapitre. La zone sensible aux SEU adressée par chacune de ces deux méthodes sera donnée dans cette section.

Parmi les 8 fenêtres de registres du processeur LEON3, seule la fenêtre qui contient les variables du programme en exécution, est considérée comme cible de la méthode CEU. Il est important de noter que quatre registres globaux seront utilisés par la procédure d'injection de fautes, et donc ne seront pas cible des SEU injectés. En effet, ces registres contiendront durant la campagne d'injection les paramètres des cibles des SEU (l'adresse cible et le masque indiquant le bit à perturber). En total, la méthode CEU peut injecter des fautes dans 28 registres de la fenêtre de registre considérée et si toute la zone sensible est considérée, la méthode CEU peut injecter de fautes dans 131 registres.

Les cache d'instructions et des données, accessible par la méthode CEU, sont tous deux configurées ayant chacune 1Kb de capacité et directement mappée. La zone cache sensible face aux SEU qui sera considérée dans les expériences d'injection de fautes est de 2Kx32 bits.

Le LEON3 contient deux compteurs de programme, PC (Program Counter) et nPC (next Program Counter). Si l'un de ces deux sera perturbé, le résultat peut être une perte de séquence ou un

résultat erroné. Ces deux registres sont localisés dans les registres locaux "l1" et "l2" de la fenêtre de registres de la routine d'interruption et sont ciblés par la méthode CEU.

Dans le cas de l'application de la méthode NETFI, l'analyse de la netlist montre que la zone sensible globale est constituée de 18983 flip-flops, non accessible via le jeux d'instructions, en addition aux cache d'instruction et de données, et aux fenêtres de registres.

Conclusion, dans le cas de la méthode CEU, les registres cibles sont les 2Kx32 bits de cache, 131 registres de la fenêtre de registres, le PC et le nPC ce qui fait un total de 2078 registres de 32 bits soit 69920 bits. NETFI cible 18983 flip-flops de contrôle et du pipeline, 2Kx32 bits des caches, et 135x32 bits des fenêtres de registres, soit 88839 bits. Donc NETFI a accès à 18938 bits de plus que la CEU (environ 21,3%).

4.3 Plateforme utilisée pour l'injection de fautes

Le testeur ASTERICS a été utilisé comme plateforme hardware pour l'injection de fautes, le processeur LEON3 étant implémenté sur son *Chipset FPGA*. Un contrôleur mémoire commun permettant l'accès au SRAM du testeur par le PC et le LEON3 est encore implémenté dans le *Chipset FPGA*. La figure 4.5 montre un diagramme de l'installation du système.

Le superviseur dans le C*hipset FPGA* a pour rôle de sauvegarder dans des registres internes l'instant de l'occurrence des SEUs, l'adresse du registre cible, et le masque des bits (le bit à perturber). Il lance l'exécution du LEON3, active le signal d'interruption dans le cas de CEU ou le signal injection dans le cas de NETFI, détecte la fin de l'application, envoie les résultats à l'ordinateur pour la comparaison avec les résultats de référence et fait face aux timeouts.

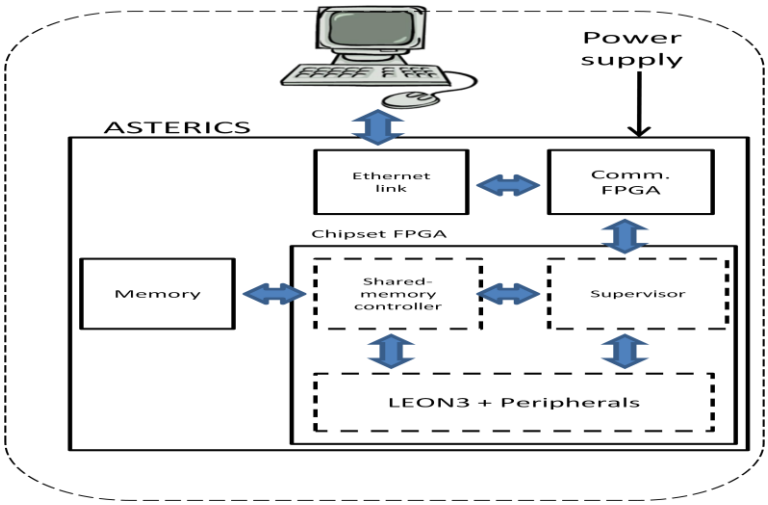


Figure 4.5: Installation du système PC-ASTERICS-LEON3

Il faut distinguer trois types de perte de séquence ou timeout:

- Timeout de démarrage: quand la séquence de démarrage ne se termine pas. Alors il faut relancer le programme.
- Timeout de ASTERICS: quand l'application exécutée par le processeur LEON3 subit une perte de s.
- Timeout du PC: quand ASTERICS n'a pas de réponse dû a un bug dans le superviseur ou une perte de connexion.

Il faut encore savoir quand l'exécution de l'application se termine, puisque la faute doit être injectée entre la fin du démarrage et la fin attendue de l'application voir figure 4.6.

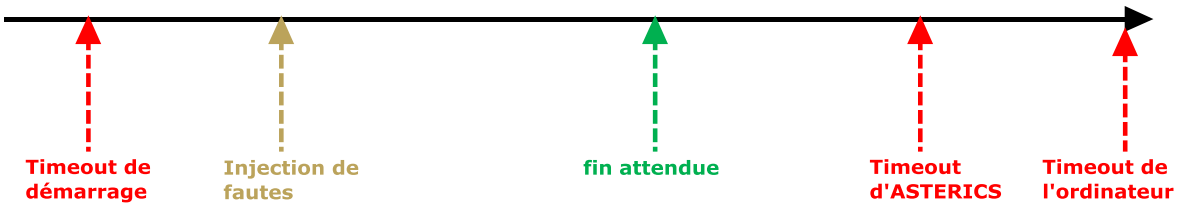


Figure 4.6 : Axe de temps du système d'injection de fautes

4.4 Campagnes d'injection de fautes réalisées avec la méthode CEU

Plusieurs sessions d'injections de fautes, via CEU, sur différentes versions de l'application ont été menées. Le but de ces expérimentations était l'étude de la capacité de tolérance aux fautes de type SEUs de l'algorithme self-convergent et l'analyse des zones et des cellules mémoires les plus critiques afin d'appliquer les modifications du cas permettant d'assurer une meilleure tolérance aux fautes.

4.4.1 Résultats préliminaires

Etant donné le code source du programme self-convergent, la robustesse intrinsèque de cet algorithme en ce qui concerne les soft-erreurs résultant du phénomène SEU a été étudiée. Pour ce but, plusieurs campagnes d'injection de fautes ciblant seulement les 28 registres de la fenêtre de registres contenant les variables du programme ont été réalisées. La limite d'exécution (timeout limit) varie d'une session à une autre, étant respectivement 1.5, 5, 8 et 16 fois le temps de la fin attendue. Les résultats obtenus confirment la capacité de cet algorithme à fournir des résultats corrects en présence de défauts affectant les données et les registres utilisées par l'algorithme. En plus, il est montré que la durée d'exécution peut cacher des erreurs qui ont été considérées comme timeouts dans des campagnes ayant un 'timeout limit' plus petit. Le tableau 4.1 montre les résultats de l'injection de fautes en

fonction du temps de simulation de l'algorithme. Les types d'erreurs considérées lors de ces campagnes sont les timeouts (si le temps d'exécution de l'application est plus grand que le timeout d'ASTERICS) et les résultats erronés (si les résultats issus de l'exécution de l'application ne correspond pas aux résultats de référence). Les injections qui n'ont pas d'effet sur les résultats sont les SEUs qui affectent soit les bits non utilisés par l'application après le temps de l'injection de la faute, soit les bits non liés aux variables qui affectent les résultats.

Le tableau 4.1 montre qu'à la durée d'exécution égale à 5 fois la fin attendue, les taux d'erreurs et des timeouts deviennent stables. En d'autres mots, si on prolonge davantage la durée d'exécution de l'application, ceci ne va pas avoir des effets remarquables sur les résultats. Il faut mentionner que le taux d'injection de fautes dans le cas de l'algorithme self-convergent est égale à 1SEU/2 Seconde.

Tableau 4.1: Résultats de l'injection de fautes sur une application self-convergente

Test	# des fautes injectées	Erreurs dans les résultats	Timeout	Durée d'exécution
1	130577	204 (0.15%)	32143 (24.61%)	1,5
2	199550	324 (0.16%)	49478 (24.79%)	1,5
3	15068	1709 (11.34%)	992 (6.58%)	5
4	14264	1614 (11.31%)	900 (6.30%)	8
5	8007	887 (11.07%)	508 (6.34%)	16

L'analyse approfondie des résultats sur les registres contenant les variables du programme self-convergent met en évidence la sensibilité de chaque variable au cours de ces campagnes d'injection de fautes. Dans le tableau 4.2 sont indiqués les types d'erreurs observées sur chacune des variables lors de cette campagne, ainsi que si elles peuvent être récupérables ou non. En général, les variables qui peuvent être durcies sont celles qui ne sont pas directement liées ni aux entrées ni aux sorties de l'algorithme, ceci pour éviter que le système ne commence pas ses calculs avec des données erronées, ni qu'il enregistre des sorties incorrectes dans la mémoire cache des données du processeur.

Tableau 4.2: Sensitivité des variables du programme self-convergent

Variable	Erreurs observées	Récupérable
I	Timeouts	Oui
J	Timeouts	Oui
M	Erreurs et timeouts	Oui
D	Erreurs et timeouts	Non
T	Erreurs et timeouts	Non
B	Timeouts	Oui
C	Timeouts	Oui

Les timeouts peuvent toujours être détectés par un "watchdog timer", tandis-que les erreurs qui affectent les données (les matrices T et D) ne peuvent pas être corrigées par l'algorithme.

Dans notre cas, pour N = 16, les entrées T représentent 256 entiers et les sortis D représentent 16 entiers. Ils en existent 5 entiers de plus qui sont les variables du programme. Les erreurs étant injectées d'une façon aléatoire dans le temps et le lieu, la plupart entre eux vont toucher les matrices T et D, ce qui explique les résultats erronés. Le tableau 4.1 montre que le taux d'erreurs est plus grand que celui prévu qui est égale à 11%.

4.4.2 Modifications du logiciel

La self-convergence est un domaine qui ne considère pas les soft-erreurs puisqu'il est implémenté en plus haut niveau et ne traite pas la partie matérielle comme la fenêtre des registres où les variables sont enregistrés. Pour avoir une meilleure tolérance aux fautes, une première modification sur le code source de l'algorithme self-convergent a été effectuée. Un opérateur modulo « % » supplémentaire est ajouté avec chaque appel des entrées et des sorties, pour garantir que l'adressage sera toujours dans la bonne zone mémoire, donc la probabilité d'avoir des entrées erronées ou d'écrire dans des zones mémoires inattendus sera presque nulle. Un exemple de l'utilisation de cet opérateur est donné par le remplacement de la douzième ligne de l'algorithme décrit dans la figure 4.2 par l'instruction suivante :

*m=D[j%16]+T[((N*i)+j)%256];*

Sachant que les entrées T sont 256 variables et les sorties D sont 16 variables, la zone d'adressage des deux matrices est de l'ordre de leurs dimensions.

Une deuxième modification consiste à attribuer à chaque variable du programme un registre local dans la fenêtre des registres, ceci en utilisant l'instruction suivante:

register unsigned int variable asm ("register name");

De cette façon, on réduit au minimum possible les ressources utilisées par la fenêtre de registres considérées. Chacune des 5 variables du programme reste dans un registre unique tout le long de la durée d'exécution. Les registres choisis sont les registres locaux L0 à L4.

Pour éviter les timeouts produits par les SEUs qui touchent les variables b et c, on a modifié ces deux variables pour qu'elles soient chacune un entier de 32-bits aléatoires au lieu d'être initialisées à un seul bit «1». Dans cette étude, on a choisi leur valeur égale à "0xFA7F". Ces variables ne peuvent maintenant pas affecter la boucle car aucun basculement d'un seul bit ne pourra rendre leurs contenus égaux à zéro en garantissant donc qu'on ne quitte pas la boucle. Une campagne d'injection de fautes a été réalisée pour mettre en évidence l'efficacité de l'algorithme self-convergent lorsque le LEON3 exécute le code modifié. La durée d'exécution a été choisie égale à 5 fois la fin attendue, parce que, comme déjà indiqué dans le tableau 4.1, à cette limite, le taux d'erreurs et des timeouts se stabilisent. Le tableau 4.3 présente les résultats de la

campagne d'injection de fautes réalisée avec l'algorithme modifié. L'analyse des résultats montre une diminution: de 11,34% à 4,45 % pour les erreurs et de 6.58% à 2,6% pour les timeouts. L'algorithme self-convergent est capable de détecter et de corriger 37,15% des fautes injectées, les modifications au niveau logiciel apportant un gain total de 10.87% (erreurs + timeouts).

Tableau 4.3 : Résultats de l'injection de fautes sur le programme modifié

# Injections	# Erreurs	# Timeouts	# Convergences
8000	356 (4.45%)	208 (2.6%)	2972 (37.15%)

Une campagne d'injection des fautes ciblant toute la zone sensible accessible par la méthode CEU, 69920 bits, a été réalisée. Les résultats issus de cette campagne sont donnés dans le tableau 4.4.

Tableau 4.4 : Résultats de l'injection de fautes avec CEU sur toute la zone sensible du LEON3

Zone sensible	#injection	#erreurs	#timeouts
69920 bits	88410	2088 (2.36%)	1345(1.52%)

88410 injections ont été réalisées avec la méthode CEU ciblant toute la partie sensible du processeur LEON3, accessible via le jeu d'instruction. Le taux d'erreurs global (erreurs + timeouts) a été 3,88%.

4.5 Campagnes d'injection de fautes réalisées par la méthode NETFI

La méthode NETFI a été utilisée pour réaliser une campagne d'injection de fautes sur la zone sensible du processeur LEON3 lorsqu'il exécute la version modifiée de l'algorithme *self-convergent*. La durée d'exécution est choisie d'être égale à 5 fois la durée de la fin attendue, car comme déjà montré dans le tableau 4.1, à cette limite les taux d'erreurs et des timeouts se stabilisent. Les résultats de la campagne d'injection de fautes est donnée par le tableau 4.5.

Tableau 4.5 : Résultats de l'injection de fautes avec NETFI sur toute la zone sensible du LEON3

Zone sensible	#injection	#erreurs	#timeouts
88839 bits	88566	2152 (2.43%)	2556(2.88%)

88566 injections ont été réalisées avec la méthode NETFI ciblant toute la partie sensible du processeur LEON3. Le taux d'erreurs global (erreurs + timeouts) a été 5,31 %.

4.6 Confrontation des résultats NETFI et CEU

La comparaison des taux d'erreurs globaux (erreurs + timeouts) issus des injections de fautes par les méthodes NETFI et CEU est donnée par la figure 4.7.

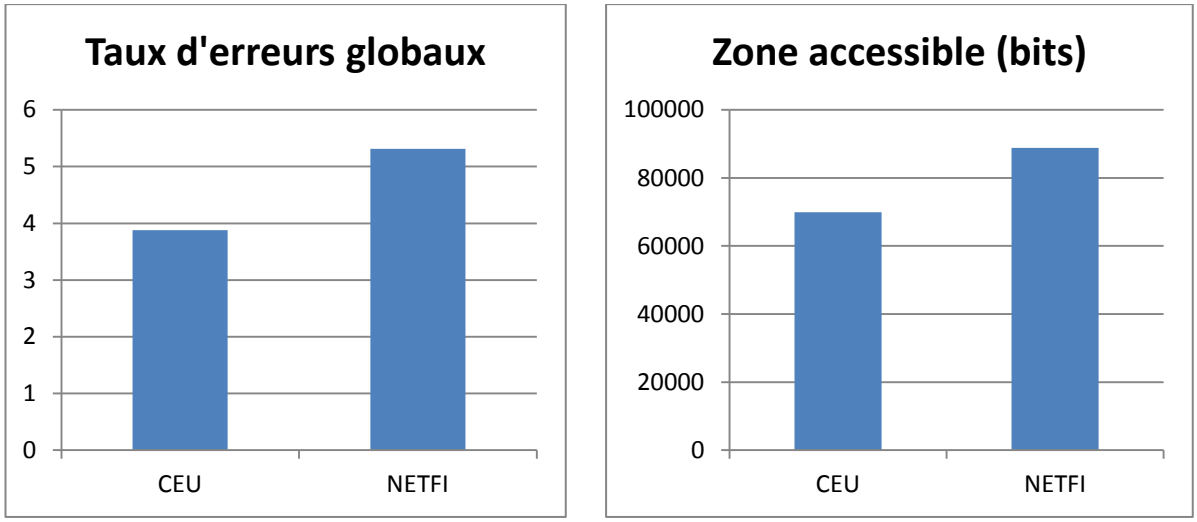


Figure 4.7: Comparaison des taux d'erreurs globaux issus de DFI et de CEU

La différence de 1,43 % entre les taux d'erreurs globaux détectés lors des deux campagnes d'injection de fautes montre que la méthode CEU, peut rater des erreurs ou des timeouts qui sont les conséquences de l'injection d'une faute dans la zone non-accessible via le jeu d'instruction tels les registres de pipeline. La surface sensible accéder par la méthode NETFI étant plus grande, plus précise en comparaison à celle de l'application finale contrairement à la méthode CEU.

Les résultats des campagnes d'injections de fautes avec la méthode NETFI sur la zone sensible non accessible via le jeux d'instruction qui représente 21,3 % de la totalité des cellules mémoires sont montrés dans le tableau 4.6.

Tableau 4.6 : Résultats de l'injection de fautes sur la zone sensible du LEON3 non-accessible via CEU

Zone sensible (bits)	# injections	# d'erreurs	# timeouts
18938	2519	42 (1,66 %)	58 (2,3%)

2519 injections ont été effectuées avec la méthode NETFI sur la zone du LEON3 non-accessible via la méthode CEU. L'analyse de cette zone montre que la plupart des cellules mémoires (15615 cellules) correspondent aux périphériques du LEON3 qui ne sont pas utilisée par le programme *self-convergent* donc les fautes injectées dans ces cellules n'ont aucun effet. Le taux d'erreurs global dans cette zone mémoire, non accessible via la méthode CEU, est de 3,96%.

Une erreurs de type SEFI (Single Event Function Interrupt) a été détecté en moyen, chaque 150 fautes injectée. Ce type d'erreur requière la réinitialisation du processeur à chaque détection. Ceci limite le nombre des fautes injectées.

4.7 Conclusions

Dans ce chapitre, les deux méthodes NETFI et CEU on été utilisées pour étudier la sensibilité face aux SEU d'un processeur LEON3 exécutant un algorithme tolérant aux fautes dit *self-convergent.* La capacité de cet algorithme de tolérer les erreurs de types SEU a été mise en évidence. Des modifications au niveau software ont été faites et validées améliorant sa robustesse.

Une comparaison des résultats issus de l'application des deux méthodes étudiées a été présentée. Cette étude concerne une analyse de la surface sensible accessible par chacune de ces deux méthodes, ainsi que les taux d'erreurs globaux issus des campagnes d'injection de fautes.

Les taux d'erreurs globaux ont eu une différence d'environ 1,5%. Ceci est du au fait que la méthode NETFI peut injecter de fautes sur 20% des cellules mémoires de plus que la méthode CEU. Ces cellules mémoires sont ceux des registres du pipeline, des unités de contrôle du processeur, des registres de contrôle des périphériques configurées pour le LEON3 etc.

Les résultats d'injection de fautes ont montré que la méthode CEU peut rater 3,96% d'erreurs injectées dans 20% des cellules qui sont pas accessible. De plus, des erreurs de type SEFI ont apparu, et pour la première fois, lors de ces campagnes.

CHAPITRE 5

Conclusions générales et perspectives

Les progrès dans les technologies de fabrication des circuits intégrés se traduisent par une augmentation potentielle de leur sensibilité face aux effets de radiations présentes dans l'environnement dans lequel ils opéreront. Dans le passé, cette problématique concernait exclusivement les applications spatiales, tandis que de nos jours elle doit être prise en compte pour toute application opérant dans l'atmosphère terrestre dont les erreurs peuvent avoir des conséquences critiques. L'évaluation de la sensibilité des circuits et des applications face aux événements provoqués par les particules énergétiques présentes dans l'environnement réel est un des objectifs de cette thèse.

Pour la mise en évidence de la criticité de cette thématique, plusieurs essais ont été réalisés dans des hautes altitudes avec pour but obtenir un feedback objectif sur la sensibilité, face aux radiations atmosphériques, des circuits mémoires SRAM issus de deux générations technologiques successives. Une plateforme de test expérimentale développée à TIMA et mise-à-jour durant cette thèse, composée des mémoires SRAMs (130 nm et 90 nm), a été activée lors de vols longs courriers, des ballons stratosphériques et dans des montagnes et des villes à hautes altitudes. Les événements singuliers et multiples détectés par cette plateforme, ont été confrontés à ceux prédits par le modèle MUSCA SEP^3/ADDICT développé à l'ONERA.

La tendance actuelle favorisant l'utilisation des circuits commerciaux pour les applications spatiales et avioniques impose la validation de leur sensibilité face aux effets de particules énergétiques. C'est dans ce contexte que la méthode de prédiction de taux d'erreurs, CEU, a été développée à TIMA dans le passé. Cette méthode permet l'injection de fautes dans des circuits de types processeurs et microprocesseurs. Dans le cadre de cette thèse, la méthode CEU a été utilisée pour étudier la sensibilité aux SEU d'un microcontrôleur PSOC exécutant une application benchmark. Ceci a mis en évidence les limitations ainsi que des désavantages de la méthode CEU, ce qui augmente le besoin à une méthode plus pertinente.

L'objet principal de cette thèse a été le développement d'une nouvelle méthode d'injection de fautes, appelé NETFI (NETlist Fault Injection), permettant d'offrir la possibilité d'estimer, la sensibilité des circuits et systèmes en technologies avancées, tôt dans le design. NETFI est une méthode automatisable et applicable sur les circuits dont on dispose de leur code RTL implémentés sur un FPGA. Elle est basée sur la manipulation de la netlist du circuit cible via la modification de la librairie *built-in* de Xilinx pour permettre l'injection de fautes principalement de types SEU, SET mais extensible aussi a des fautes de type Stuck_at. La plateforme ASTERICS de TIMA a été le support matériel pour la réalisation des essais requis pour valider cette méthode.

Une première campagne d'injection de fautes, avec NETFI, a été réalisée sur un processeur complexe, le LEON2, exécutant un benchmark *bubble_sort*, ceci dans le but d'illustrer la méthode ainsi que son utilisation afin d'étudier la sensibilité face aux soft-erreurs du processeur considéré.

Le microcontrôleur 80C51 d'Intel a été la deuxième cible de la méthode NETFI. Ceci pour comparer les résultats de la prédiction issus de NETFI à ceux issus des tests sous faisceaux de particules et qui étaient disponibles dans la littérature, pour le même programme benchmark, multiplication des matrices, exécuté par le microcontrôleur considéré. La confrontation des mesures et des prédictions a permit la mise en évidence de la pertinence des résultats obtenus par la méthode NETFI. Une comparaison avec la méthode CEU a été faite aussi dans le but de montrer les zones accessibles par chaque méthode ainsi que d'avoir un feedback sur la corrélation entre leurs résultats prédits. Dans le cas du microcontrôleur considéré, la méthode NETFI peut accéder 7% des cellules mémoires sensibles de plus que la méthode CEU. Ce faible pourcentage justifie la bonne corrélation entre les résultats prédits par chaque méthode.

L'un des buts de la méthode NETFI est la validation des designs tolérants aux fautes. Le troisième circuit ciblé a été un Réseau de Neurones Artificiels de type Hopfield (RNH). Ce circuit, qui a été utilisé dans le passé dans des applications spatiales, fut choisi pour sa capacité intrinsèque à tolérer des erreurs qui peuvent se produire durant le calcul. Deux versions de ce RNH ont été conçues, au cours de cette thèse, et implémentées sur un FPGA. Les campagnes d'injections de fautes, de types SEU, SET et Stuck_at, faites utilisant la méthode NETFI ont montré la robustesse de l'implémentation de la version RNH tolérante aux fautes : plus robuste d'un facteur 100 que la version originale.

Une étude sur un algorithme tolérant aux fautes dit *self-convergent* a été aussi réalisée dans le but de comparer les méthodes NETFI et CEU, lorsque le circuit cible est un processeur complexe: le LEON3. Des campagnes de test ont été réalisées avec la méthode CEU ayant pour but d'identifier les potentiels tallons d'Achilles de l'algorithme *self-convergent*. Les résultats obtenus mettent en évidence la capacité de l'algorithme original à tolérer les SEUs dus aux effets des radiations. Des modifications logicielles ont été appliquées sur le code « C » de l'algorithme considéré résultant en une atténuation significative (environ 3 fois) de la sensibilité face aux soft-erreurs du LEON3 exécutant l'algorithme considéré. La comparaison des résultats d'injections de fautes, effectuées à l'aide de NETFI et CEU sur les zones sensibles accessibles par chaque méthode, a montrée que la méthode NETFI peut toucher 20% environ des cellules mémoires de plus que la méthode CEU. Des campagnes d'injection de fautes ciblant seulement cette zone résultent en un taux d'erreurs global (erreurs + timeouts) d'environ 4%. Ceci montre que la méthode CEU peut rater des erreurs critiques dans une zone non-accessible. Un type d'erreur qui apparait lors de ces campagnes est le SEFI (Single Event Functional Interrupt), qui requière à chaque fois la réinitialisation du processeur.

Parmi les perspectives de ces travaux, peuvent être mentionnées:

- La conception d'une nouvelle plateforme de test pour des mémoires 3D ou des mémoires SRAM issues de technologies plus avancées à 65 nm. Ceci pour étudier l'effet des radiations présentes dans l'environnement réel sur l'avancement de la technologie.

- Réalisations des campagnes de test sous faisceaux de particules pour valider les résultats issus des injections de fautes par NETFI sur le processeur LEON2 exécutant les programmes benchmarks considérés.

- Le développement d'une application GUI (Graphical User Interface) pour faciliter l'utilisation et la mise à jour de la méthode NETFI.

A titre de conclure, il est prévu développé et implémenté un large RNH dédié à une application spatiale pour l’embarquer dans la charge utile d’un satellite scientifique.

Annexe A. Les modules de Xilinx modifiées

```
module FD_mod (
input inj,
input D,
output Q,
input C);
 parameter INIT = 1'b0;
wire Din;
assign Din = (inj) ? !D : D;
FD uut_FDC (.Q(Q), .C(C), .D(Din));
endmodule
**********************************************************************************

module FDC_mod (inj,Q, C, CLR, D);
 parameter INIT = 1'b0;
   output Q;
   input  C, CLR, D, inj;
   wire Din;
   FDC uut(.Q(Q),.C(C),.CLR(CLR),.D(Din));
   assign Din = (inj) ? !D : D;
endmodule
**********************************************************************************

module FDCE_mod (inj, Q, C, CE, CLR, D);
parameter INIT = 1'b0;
   output Q;
   input  inj, C, CE, CLR, D;
   wire Q;
   wire q_out;
   wire Din;

        FDCE uut(.Q(Q),.C(C),.CE(CE||inj),.CLR(CLR),.D(Din));
        assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D : D;
endmodule
**********************************************************************************
```

```
module FDCP_mod (inj, Q, C, CLR, D, PRE);

 parameter INIT = 1'b0;
  output Q;
  input  inj, C, CLR, D, PRE;

  wire Q_n;
  wire Din;
  wire En;
 assign Q_n = Q;
 assign Din = (!inj) ? D :  !D ;
 FDCP uut_FDE(.CLR(CLR),.PRE(PRE), .Q(Q), .D(Din), .C(C) );
endmodule
**********************************************************************************

    module FDE_mod (
    input inj,
          output Q,
          input D,
          input C,
          input CE);

          wire Din;
          FDE uut (.Q(Q),.D(Din),.C(C),.CE(CE||inj));
          assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D : Q;
 endmodule
**********************************************************************************

module FDP_mod (inj,Q, C, D, PRE);
    parameter INIT = 1'b0;
    output Q;
    wire Din;
    input inj, C, D, PRE;
    FDP uut(.C(C),.D(Din),.PRE(PRE),.Q(Q));
      assign Din = (inj) ? !D : D;
endmodule
**********************************************************************************
```

```
module FDPE_mod (inj,Q, C, CE, D, PRE);
   parameter INIT = 1'b0;
   output Q;
   input  inj, C, CE, D, PRE;
  wire Din;
   FDPE uut (.Q(Q),.C(C),.CE(CE),.D(Din),.PRE(PRE));
   assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D :  Q;
endmodule
**********************************************************************************

module FDR_mod (inj, Q, C, D, R);
   parameter INIT = 1'b0;
   output Q;
   input  inj, C, D, R;
   wire Q;
   reg q_out;
   assign Q = q_out;
   always @(posedge C )
   begin
 if(!inj)
      if (R)
      q_out <=  0;
       else
      q_out <=  D;
   else
      if (R)
      q_out <=  1;
       else
      q_out <=  !D;
    end
endmodule
**********************************************************************************

module FDRE_mod (inj, Q, C, CE, D, R);
 parameter INIT = 1'b0;
   output Q;
   input  C, CE, D, R, inj;
   wire Q;
```

```
   reg q_out;
  assign Q = q_out;
         always @(posedge C )
   begin
      if (R && !inj)
            q_out <=  0;
           else if (R && inj) q_out <= 1;
           else if (CE && !inj)
            q_out <=  D;
           else if(CE && inj) q_out <= !D;
           else if (!CE && inj) q_out <= !q_out;
         end
endmodule
*********************************************************************************

module FDRSE_mod (inj, Q, C, CE, D, R, S);
   parameter INIT = 1'b0;
   output Q;
   input  inj, C, CE, D, R, S;
   wire Q;
   reg q_out;
   assign Q = q_out;
   always @(posedge C )
   begin
   if(!inj)
      if (R)
      q_out <=  0;
      else if (S)
      q_out <=  1;
      else if (CE)
      q_out <=  D;
   else
      if (R)        q_out <=  1;
      else if (S)        q_out <=  0;
      else if (CE)        q_out <=  !D;
     else if(!CE)          q_out <= !q_out;
   end
endmodule
*********************************************************************************
```

```
module FDS_mod (inj, Q, C, D, S);
   parameter INIT = 1'b1;
   output Q;
   input  inj, C, D, S;
   wire Q;
   reg q_out;
   assign Q = q_out;
   always @(posedge C )
    begin
    if(!inj)
         if (S)
         q_out <=  1;
         else
         q_out <=  D;
    else
       if(S)
          q_out <= 0;
       else
          q_out <= !D;
    end
endmodule
**********************************************************************************

`timescale  1 ps / 1 ps
module RAM16X1D_mod (inj, DPO, SPO, A0, A1, A2, A3, D, DPRA0, DPRA1, DPRA2, DPRA3,
WCLK, WE);
   parameter INIT = 16'h0000;
   output DPO, SPO;
   input  A0, A1, A2, A3, D, DPRA0, DPRA1, DPRA2, DPRA3, WCLK, WE;
   input [15:0] inj;
   reg  [15:0] mem;
   wire [3:0] adr;
   assign adr = {A3, A2, A1, A0};
   assign SPO = mem[adr];
   assign DPO = mem[{DPRA3, DPRA2, DPRA1, DPRA0}];
   initial
      mem = INIT;
   always @(posedge WCLK)
```

```
    begin
        if (WE == 1'b1) mem[adr] <= #100 D;
        case(inj)
        16'h0001: mem[0] <= !mem[0];
        16'h0002: mem[1] <= !mem[1];
        16'h0004: mem[2] <= !mem[2];
        16'h0008: mem[3] <= !mem[3];
        16'h0010: mem[4] <= !mem[4];
        16'h0020: mem[5] <= !mem[5];
        16'h0040: mem[6] <= !mem[6];
        16'h0080: mem[7] <= !mem[7];
        16'h0100: mem[8] <= !mem[8];
        16'h0200: mem[9] <= !mem[9];
        16'h0400: mem[10] <= !mem[10];
        16'h0800: mem[11] <= !mem[11];
        16'h1000: mem[12] <= !mem[12];
        16'h2000: mem[13] <= !mem[13];
        16'h4000: mem[14] <= !mem[14];
        16'h8000: mem[15] <= !mem[15];
        default: mem <= mem;
        endcase
      end
endmodule
*********************************************************************************

`timescale  1 ps / 1 ps
module RAM32X1D_mod (inj, DPO, SPO, A0, A1, A2, A3, A4, D, DPRA0, DPRA1, DPRA2, DPRA3,
DPRA4, WCLK, WE);
  parameter INIT = 32'h00000000;
  output DPO, SPO;
  input  A0, A1, A2, A3, A4, D, DPRA0, DPRA1, DPRA2, DPRA3, DPRA4, WCLK, WE;
  input [31:0] inj;
  reg  [31:0] mem;
  wire [4:0] adr;
  assign adr = {A4, A3, A2, A1, A0};
  assign SPO = mem[adr];
  assign DPO = mem[{DPRA4, DPRA3, DPRA2, DPRA1, DPRA0}];
  initial
    mem = INIT;
  always @(posedge WCLK)
```

```
    begin
      if (WE == 1'b1)
           mem[adr] <= #100 D;
           case(inj):
           32'h00000001: mem[0] <= !mem[0];
           32'h00000002: mem[1] <= !mem[1];
           32'h00000004: mem[2] <= !mem[2];
           32'h00000008: mem[3] <= !mem[3];
           32'h00000010: mem[4] <= !mem[4];
           32'h00000020: mem[5] <= !mem[5];
           32'h00000040: mem[6] <= !mem[6];
           32'h00000080: mem[7] <= !mem[7];
           32'h00000100: mem[8] <= !mem[8];
           32'h00000200: mem[9] <= !mem[9];
           32'h00000400: mem[10] <= !mem[10];
           32'h00000800: mem[11] <= !mem[11];
           32'h00001000: mem[12] <= !mem[12];
           32'h00002000: mem[13] <= !mem[13];
           32'h00004000: mem[14] <= !mem[14];
           32'h00008000: mem[15] <= !mem[15];
           32'h00010000: mem[16] <= !mem[16];
           32'h00020000: mem[17] <= !mem[17];
           32'h00040000: mem[18] <= !mem[18];
           32'h00080000: mem[19] <= !mem[19];
           32'h00100000: mem[20] <= !mem[20];
           32'h00200000: mem[21] <= !mem[21];
           32'h00400000: mem[22] <= !mem[22];
           32'h00800000: mem[23] <= !mem[23];
           32'h01000000: mem[24] <= !mem[24];
           32'h02000000: mem[25] <= !mem[25];
           32'h04000000: mem[26] <= !mem[26];
           32'h08000000: mem[27] <= !mem[27];
           32'h10000000: mem[28] <= !mem[28];
           32'h20000000: mem[29] <= !mem[29];
           32'h40000000: mem[30] <= !mem[30];
           32'h80000000: mem[31] <= !mem[31];
           default: mem <= mem;
           endcase
endmodule
```

****

```
`timescale  1 ps / 1 ps
module RAM64X1D_mod (inj, DPO, SPO, A0, A1, A2, A3, A4, A5, D, DPRA0, DPRA1, DPRA2,
DPRA3, DPRA4, DPRA5, WCLK, WE);
   parameter INIT = 64'h0000000000000000;
   output DPO, SPO;
   input  A0, A1, A2, A3, A4, A5, D, DPRA0, DPRA1, DPRA2, DPRA3, DPRA4, DPRA5, WCLK, WE;
   input [63:0] inj;
   reg [63:0] mem;
   wire [5:0] adr;
   assign adr = {A5, A4, A3, A2, A1, A0};
   assign SPO = mem[adr];
   assign DPO = mem[{DPRA5, DPRA4, DPRA3, DPRA2, DPRA1, DPRA0}];
   initial
      mem = INIT;
   always @(posedge WCLK)
         begin
         if (WE == 1'b1)
            mem[adr] <= #100 D;
         case(inj):
         64'h0000000000000001: mem[0] <= !mem[0];
         64'h0000000000000002: mem[1] <= !mem[1];
         64'h0000000000000004: mem[2] <= !mem[2];
         64'h0000000000000008: mem[3] <= !mem[3];
         64'h0000000000000010: mem[4] <= !mem[4];
         64'h0000000000000020: mem[5] <= !mem[5];
         64'h0000000000000040: mem[6] <= !mem[6];
         64'h0000000000000080: mem[7] <= !mem[7];
         64'h0000000000000100: mem[8] <= !mem[8];
         64'h0000000000000200: mem[9] <= !mem[9];
         64'h0000000000000400: mem[10] <= !mem[10];
         64'h0000000000000800: mem[11] <= !mem[11];
         64'h0000000000001000: mem[12] <= !mem[12];
         64'h0000000000002000: mem[13] <= !mem[13];
         64'h0000000000004000: mem[14] <= !mem[14];
         64'h0000000000008000: mem[15] <= !mem[15];
         64'h0000000000010000: mem[16] <= !mem[16];
         64'h0000000000020000: mem[17] <= !mem[17];
```

```
64'h0000000000040000: mem[18] <= !mem[18];
64'h0000000000080000: mem[19] <= !mem[19];
64'h0000000000100000: mem[20] <= !mem[20];
64'h0000000000200000: mem[21] <= !mem[21];
64'h0000000000400000: mem[22] <= !mem[22];
64'h0000000000800000: mem[23] <= !mem[23];
64'h0000000001000000: mem[24] <= !mem[24];
64'h0000000002000000: mem[25] <= !mem[25];
64'h0000000004000000: mem[26] <= !mem[26];
64'h0000000008000000: mem[27] <= !mem[27];
64'h0000000010000000: mem[28] <= !mem[28];
64'h0000000020000000: mem[29] <= !mem[29];
64'h0000000040000000: mem[30] <= !mem[30];
64'h0000000080000000: mem[31] <= !mem[31];
64'h0000000100000000: mem[32] <= !mem[32];
64'h0000000200000000: mem[33] <= !mem[33];
64'h0000000400000000: mem[34] <= !mem[34];
64'h0000000800000000: mem[35] <= !mem[35];
64'h0000001000000000: mem[36] <= !mem[36];
64'h0000002000000000: mem[37] <= !mem[37];
64'h0000004000000000: mem[38] <= !mem[38];
64'h0000008000000000: mem[39] <= !mem[39];
64'h0000010000000000: mem[40] <= !mem[40];
64'h0000020000000000: mem[41] <= !mem[41];
64'h0000040000000000: mem[42] <= !mem[42];
64'h0000080000000000: mem[43] <= !mem[43];
64'h0000100000000000: mem[44] <= !mem[44];
64'h0000200000000000: mem[45] <= !mem[45];
64'h0000400000000000: mem[46] <= !mem[46];
64'h0000800000000000: mem[47] <= !mem[47];
64'h0001000000000000: mem[48] <= !mem[48];
64'h0002000000000000: mem[49] <= !mem[49];
64'h0004000000000000: mem[50] <= !mem[50];
64'h0008000000000000: mem[51] <= !mem[51];
64'h0010000000000000: mem[52] <= !mem[52];
64'h0020000000000000: mem[53] <= !mem[53];
64'h0040000000000000: mem[54] <= !mem[54];
64'h0080000000000000: mem[55] <= !mem[55];
64'h0100000000000000: mem[56] <= !mem[56];
```

```
        64'h0200000000000000: mem[57] <= !mem[57];
        64'h0400000000000000: mem[58] <= !mem[58];
        64'h0800000000000000: mem[59] <= !mem[59];
        64'h1000000000000000: mem[60] <= !mem[60];
        64'h2000000000000000: mem[61] <= !mem[61];
        64'h4000000000000000: mem[62] <= !mem[62];
        64'h8000000000000000: mem[63] <= !mem[63];
        default: mem <= mem;
        endcase
        end
endmodule
*********************************************************************************

module INV_mod (inj, O, I);
 output O;
 input  inj, I;
wire O1;
not N1 (O1, I);
assign O = (!inj) ? O1 : 1;//0;//!O1;      1 for stuck_at_1, 0 for stuck_at_0, !O1 for SET
endmodule
*********************************************************************************

module LUT2_mod (inj, O, I0, I1);
 parameter INIT = 4'h0;
 input I0, I1, inj;
 output O;
 reg  O1;
 wire [1:0] s;
 assign s = {I1, I0};
assign O = (!inj) ? O1 : 1;//0;//!O1;      1 for stuck_at_1, 0 for stuck_at_0, !O1 for SET
   always @(s)
     if ((s[1]^s[0] ==1) || (s[1]^s[0] ==0))
       O1 = INIT[s];
     else if ((INIT[0] == INIT[1]) && (INIT[2] == INIT[3]) && (INIT[0] == INIT[2]))
       O1 = INIT[0];
     else if ((s[1] == 0) && (INIT[0] == INIT[1]))
       O1 = INIT[0];
     else if ((s[1] == 1) && (INIT[2] == INIT[3]))
       O1 = INIT[2];
     else if ((s[0] == 0) && (INIT[0] == INIT[2]))
```

```
        O1 = INIT[0];
      else if ((s[0] == 1) && (INIT[1] == INIT[3]))
        O1 = INIT[1];
      else
        O1 = 1'bx;
endmodule
**********************************************************************************

module LUT3_mod (inj, O, I0, I1, I2);
 parameter INIT = 8'h00;
 input I0, I1, I2, inj;
 output O;
 reg O1;
 reg tmp;
assign O = (!inj) ? O1 : 1;//0;// !O1;     1 for stuck_at_1, 0 for stuck_at_0, !O1 for SET
 always @(  I2 or  I1 or  I0 )  begin
    tmp =  I0 ^ I1  ^ I2;
    if ( tmp == 0 || tmp == 1)
        O1 = INIT[{I2, I1, I0}];
    else
        O1 = lut3_mux4 ( {1'b0, 1'b0, lut3_mux4 (INIT[7:4], {I1, I0}),
lut3_mux4 (INIT[3:0], {I1, I0}) }, {1'b0, I2});    end
  function lut3_mux4;
  input [3:0] d;
  input [1:0] s;
  begin
     if ((s[1]^s[0] ==1) || (s[1]^s[0] ==0))
        lut3_mux4 = d[s];
      else if ((d[0] === d[1]) && (d[2] === d[3]) && (d[0] === d[2])) lut3_mux4 = d[0];
            else if ((s[1] == 0) && (d[0] === d[1])) lut3_mux4 = d[0];
                 else if ((s[1] == 1) && (d[2] === d[3])) lut3_mux4 = d[2];
                      else if ((s[0] == 0) && (d[0] === d[2])) lut3_mux4 = d[0];
                           else if ((s[0] == 1) && (d[1] === d[3])) lut3_mux4 = d[1];
                                else  lut3_mux4 = 1'bx;
   end
   endfunction
endmodule
**********************************************************************************
```

```
module LUT4_mod (inj, O, I0, I1, I2, I3);
 parameter INIT = 16'h0000;
 input inj, I0, I1, I2, I3;
 output O;
 reg O1;
 reg tmp;
assign O = (!inj) ? O1 : 1;//0;//!O1;
always @(  I3 or  I2 or  I1 or  I0 )  begin
    tmp =  I0 ^ I1  ^ I2 ^ I3;
  if ( tmp == 0 || tmp == 1)
     O1 = INIT[{I3, I2, I1, I0}];
  else
    O1 =  lut4_mux4 ( {lut4_mux4 ( INIT[15:12], {I1, I0}),
                  lut4_mux4 ( INIT[11:8], {I1, I0}),
                  lut4_mux4 ( INIT[7:4], {I1, I0}),
                  lut4_mux4 ( INIT[3:0], {I1, I0}) }, {I3, I2});
 end
 function lut4_mux4;
 input [3:0] d;
 input [1:0] s;
 begin
     if ((s[1]^s[0] ==1) || (s[1]^s[0] ==0))
        lut4_mux4 = d[s];
      else if ((d[0] === d[1]) && (d[2] === d[3]) && (d[0] === d[2]))
        lut4_mux4 = d[0];
      else if ((s[1] == 0) && (d[0] === d[1]))
        lut4_mux4 = d[0];
      else if ((s[1] == 1) && (d[2] === d[3]))
        lut4_mux4 = d[2];
      else if ((s[0] == 0) && (d[0] === d[2]))
        lut4_mux4 = d[0];
      else if ((s[0] == 1) && (d[1] === d[3]))
        lut4_mux4 = d[1];
      else
        lut4_mux4 = 1'bx;
  end
```

```
  endfunction
endmodule
```

```
module XOR2_mod (inj, O, I0, I1);

    output O;

    input  I0, I1, inj;
wire in;

          xor X1 (in, I0, I1);
          xor X2 (O,in,inj);


endmodule

*****************************************************************************
module AND3_mod (inj, O, I0, I1, I2);

    output O;

    input  I0, I1, I2, inj;
   wire in;

    and A1 (in, I0, I1, I2);
   xor A2 (O, in, inj)


endmodule

*****************************************************************************
module SRL16_mod  (inj, Q, A0, A1, A2, A3, CLK, D);

    parameter INIT = 16'h0000;

    output Q;

    input  A0, A1, A2, A3, CLK, D;
input [15:0] inj;
```

```
    reg  [15:0] data;


    assign Q = data[{A3, A2, A1, A0}];


    always @(posedge CLK)
    begin
          {data[15:0]} <= #100 {data[14:0], D};
          case(inj)
            16'h0001: data[0] <= !data[0];
            16'h0002: data [1] <= !data[1];
            16'h0004: data [2] <= !data[2];
            16'h0008: data [3] <= !data[3];
            16'h0010: data [4] <= !data[4];
            16'h0020: data [5] <= !data[5];
            16'h0040: data [6] <= !data[6];
            16'h0080: data [7] <= !data[7];
            16'h0100: data [8] <= !data[8];
            16'h0200: data [9] <= !data[9];
            16'h0400: data [10] <= !data[10];
            16'h0800: data[11] <= !data[11];
            16'h1000: data[12] <= !data[12];
            16'h2000: data[13] <= !data[13];
            16'h4000: data[14] <= !data[14];
            16'h8000: data[15] <= !data[15];
            default: data <= data;

    end

endmodule
```

Annexe B. L'outil Modnet

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Data.SqlClient;

namespace netlistmod
{
   class Program

   {
      static void Main(string[] args)
      {
         //set the path of the netlist
         string path = @"C:\Users\Wassim Mansour\Desktop\analysis\";
         string outputpath = @"C:\Users\Wassim Mansour\Desktop\analysis\output\";
         string srcpath = @"C:\Users\Wassim Mansour\Desktop\analysis\src\";
         System.IO.Directory.CreateDirectory(srcpath);
         string[] files;
         string netlist = "leon2"; //netlist name
         string filename = "leon2"; //top module name

         string temp;
         string timescale = "";
         string netfile = System.IO.File.ReadAllText(path + netlist + ".vm");
         int i = 0;
         int j = 0;
         int k = 0;
         int m = 0;
         files = Directory.GetFiles(srcpath);
         if (files.Length < 1)
         {
```

```
        Console.Write("generating modules from netlist..."); //generating modules from netlist
        i = netfile.IndexOf("`timescale");
        j = netfile.IndexOf("module ");
        for (k = i; k < j; k++)
            timescale += netfile[k];
        do
        {
            i = netfile.IndexOf("module ");
            j = netfile.IndexOf("endmodule ");
            char[] currentmodule = new char[j];
            char[] currentmodule_n = new char[j - i];
            System.IO.StreamReader n_file = new System.IO.StreamReader(path + netlist + ".vm");
            n_file.ReadBlock(currentmodule, 0, j);
            n_file.Close();
            for (k = i; k < j; k++)
                currentmodule_n[k - i] = currentmodule[k];
            temp = new string(currentmodule);
            string currentmodule_s = new string(currentmodule_n);
            netfile = netfile.Replace(temp, "");
            i = netfile.IndexOf("endmodule ");
            j = netfile.IndexOf("*/");
            temp = "";
            for (k = i; k < j; k++)
                temp += netfile[k];
            currentmodule_s += temp + "*/";
            netfile = netfile.Replace(temp + "*/", "");
            temp = "";
            System.IO.File.WriteAllText(srcpath + secondword(currentmodule_s) + ".v", timescale +
currentmodule_s);
            System.IO.File.WriteAllText(path + netlist + ".vm", netfile);
            m++;
        } while (netfile.IndexOf("endmodule ") >= 0);
        Console.WriteLine("Done!!!");
    }
    deletefile(srcpath, "temp");
    injection_analysis(srcpath, filename, outputpath);
    Console.WriteLine("Done!!!");
    System.Console.ReadKey();
}
```

```
public static int injection_analysis(string path, string filename, string outputpath)
{
    string errtype = "SET"; //SET, RAMB
    int i = 0;
    int j = 0;
    string line = "";
    string new_file = "";
    string module_name = "";
    int mod_count;

    string analysis = "";
    int counter = 0;
    string timescale = "";
    Console.Write("reading netlist for " + filename + "...");
    System.IO.StreamReader net_file = new System.IO.StreamReader(path + filename + ".v");
    timescale = net_file.ReadLine();
    Console.WriteLine("[Done]!!");
    Console.Write("reading module name... ");
    module_name = net_file.ReadLine();
    Console.WriteLine("[Done]!!");
    Console.Write("reading module ports... ");
    string file = net_file.ReadToEnd();
    i = 0;
    j = file.IndexOf(")");
    System.IO.File.WriteAllText(path + "temp" + ".txt", file);
    char[] listport = new char[j + 4];
    net_file.Close();
    net_file = new System.IO.StreamReader(path + "temp" + ".txt");
    net_file.ReadBlock(listport, 0, j + 4);
    string module_ports = new string(listport);
    Console.WriteLine("[Done]!!");
    Console.Write("reading module port declarations... ");
    i = j + 5;
    j = file.IndexOf("wire");
    char[] decport = new char[j - i];
    net_file.ReadBlock(decport, 0, j - i);
    string module_dec = new string(decport);
    Console.WriteLine("[Done]!!");
```

```
Console.Write("reading module wires... ");
i = j;
j = file.LastIndexOf("wire ");
char[] wireport = new char[j - i];
net_file.ReadBlock(wireport, 0, j - i);
string module_wire = new string(wireport);
net_file.Close();
file = file.Replace(module_ports + module_dec + module_wire, "");
System.IO.File.WriteAllText(path + "temp" + ".txt", file);
net_file = new System.IO.StreamReader(path + "temp" + ".txt");
net_file.ReadLine();
module_wire += net_file.ReadLine() + "\n";

Console.WriteLine("[Done]!!");
//  Console.WriteLine(module_wire);
new_file = net_file.ReadToEnd();
net_file.Close();
Console.WriteLine("Analyzing submodules for " + filename + "... ");
System.IO.File.WriteAllText(path + filename + "1.txt", new_file);
System.IO.StreamReader n_file = new System.IO.StreamReader(path + filename + "1.txt");

  while ((line = n_file.ReadLine()) != null)
{

    if (line.Contains(" (") && (iscombinational(AnalyseLine(line))) && errtype == "SET") {
analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n    .inj(inj[" +
(counter).ToString() + "]) ,\n"; counter++; }

    else if (line.Contains(" (") && (issequential(AnalyseLine(line))) && errtype == "SEU") {
analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n    .inj(inj[" +
(counter).ToString() + "]) ,\n"; counter++; }

    else if (line.Contains(" (") && (isRAMB16_S36(AnalyseLine(line)) && errtype ==
"RAMB")) { analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n
.inj(inj[" + (counter).ToString() + "]) , .data_mask(data_mask), .address(address), \n"; counter++; }

    else if (line.Contains(" (") && (isRAM16X1D(AnalyseLine(line)))&& errtype == "SEU") {
analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n    .inj(inj[" + (counter
+ 15).ToString() + ":" + (counter).ToString() + "]) ,\n"; counter = counter + 16 ; }
```

```
        else if (line.Contains(" (") && (isRAM32X1D(AnalyseLine(line))) && errtype == "SEU") { analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n    .inj(inj[" + (counter + 31).ToString() + ":" + (counter).ToString() + "]) ,\n"; counter = counter + 32; }


        else if (line.Contains(" (") && (isRAM64X1D(AnalyseLine(line))) && errtype == "SEU") { analysis += line.Replace(AnalyseLine(line), AnalyseLine(line) + "_mod") + "\n    .inj(inj[" + (counter + 63).ToString() + ":" + (counter).ToString() + "]) ,\n"; counter = counter + 64; }


        else if (line.Contains(" (") && File.Exists(path + AnalyseLine(line) + ".v") && (errtype == "SEU" || errtype == "SET"))
        {
            mod_count = injection_analysis(path, AnalyseLine(line), outputpath);
            if (mod_count != 0)
            { analysis += AnalyseLine(line) + " " + AnalyseLine(line) + "_uut (\n  .inj(inj[" + (counter + mod_count - 1).ToString() + " : " + counter + "]),\n"; counter += mod_count; }
            else { analysis += AnalyseLine(line) + " " + AnalyseLine(line) + "_uut (\n"; counter += mod_count; }
        }
        else if (line.Contains(" (") && File.Exists(path + AnalyseLine(line) + ".v") && errtype == "RAMB")
        {
            mod_count = injection_analysis(path, AnalyseLine(line), outputpath);
            if (mod_count != 0)
            { analysis += AnalyseLine(line) + " " + AnalyseLine(line) + "_uut (\n  .inj(inj[" + (counter + mod_count - 1).ToString() + " : " + counter + "]),\n.data_mask(data_mask),\n.address(address),\n"; counter += mod_count; }
            else { analysis += AnalyseLine(line) + " " + AnalyseLine(line) + "_uut (\n"; counter += mod_count; }
        }
        else if (line.Contains("defparam")) analysis += line + "\n";
        else if (line.Contains("//")) analysis += line + "\n";
        else if (line.Contains(");")) analysis += line + "\n";
        else if (line.Contains(".")) analysis += line + "\n";
        else analysis += line;


    }
    n_file.Close();
    deletefile(path, filename + "1");
    deletefile(path, "temp");
    System.IO.Directory.CreateDirectory(outputpath);
    string temp1 = "";
    if (errtype == "SEU" || errtype == "SET")
```

```
        {
            if (counter != 0)
                temp1 = module_name + "\n  inj,\n" + module_ports + "\ninput [" + (counter -
1).ToString() + ": 0] inj ;\n" + module_dec + module_wire + "\n" + analysis;
            else temp1 = module_name + "\n  inj,\n" + module_ports + "\ninput  inj ;\n" + module_dec
+ module_wire + "\n" + analysis;
            System.IO.File.WriteAllText(outputpath + filename + ".v", temp1);
        }
        if(errtype == "RAMB")
        {
        if (counter != 0 )
            temp1 = module_name + "\n  inj,\ndata_mask,\naddress,\n" + module_ports + "\ninput [" +
(counter - 1).ToString() + ": 0] inj ;\ninput [35:0] data_mask;\ninput [8:0] address;\n" + module_dec
+ module_wire + "\n" + analysis;
            else temp1 = module_name + "\n  inj,\ndata_mask,\naddress,\n" + module_ports + "\ninput
inj ;\ninput [35:0] data_mask;\ninput [8:0] address;\n" + module_dec + module_wire + "\n" +
analysis;
            System.IO.File.WriteAllText(outputpath + filename + ".v", temp1);
        }
        Console.WriteLine("[Done Analysis for " + filename + "]!!");
        return counter;

    }


    public static bool iscombinational(string s)
    {
        //add all the combinational gates

      if (s == "LUT4_L" || s == "INV" || s == "LUT3" || s == "LUT2" || s == "LUT4" || s ==
"MUXF5" || s == "MUXF8" || s == "MUXF7" || s == "LUT3_L" || s == "BUFGP")

            return true;
        else return false;
    }


    public static bool issequential(string s)
    {
        //add all FFs
```

```
        if ((s == "FDD" || s == "FDE" || s == "FDC_1" || s == "FDC" || s == "FDCE" || s == "FD"
|| s == "FDPE" || s == "FDP" || s == "FDRE" || s == "FDRSE" || s == "FDR" || s == "FDS" || s
== "FDSE" || s == "FDRS")) return true;
            return false;
        }


        public static bool isRAM16X1D(string s)
        {
            if (s == "RAM16X1D")
                return true;
            else return false;
        }


        public static bool isRAM32X1D(string s)
        {
            if (s == "RAM32X1D")
                return true;
            else return false;
        }


        public static bool isRAM64X1D(string s)
        {
           if (s == "RAM64X1D")

                return true;
            else return false;
        }

        public static bool isRAMB16_S36(string s)
        {

            if (s == "RAMB16_S36") return true;
            return false;
        }

        public static string AnalyseLine(string line)
        {
```

```
        string[] words = line.Split(' ');
        return words[2];
    }
    public static string secondword(string line)
    {
        string[] words = line.Split(' ');
        return words[1];
    }


    public static void deletefile(string path, string name)
    {
        if (System.IO.File.Exists(path + name + ".txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(path + name + ".txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }
    }
  }


}
```

Annexe C. Publications pendant la thèse

Journaux

W. Mansour, R. Ayoubi, H. Ziade, R. Velazco, W. El Falou, An Optimal Implementation on FPGA of a Hopfield Neural Network, Advances in Artificial Neural Systems, Volume 2011 (2011), Article 189368, 9 pages, doi: 10.1155/2011/189368, Hindawi Publishing Corporation, 2011.

L. Artola, R. Velazco, G. Hubert, S. Duzellier, T. Nuns, B. Guerard, P. Peronnard, W. Mansour, F. Pancher, F. Bezerra, In Flight SEU/MCU Sensitivity of Commercial Nanometric SRAMs: Operational Estimations, IEEE Transactions on Nuclear Science, vol. 58, pp 2644-2651, Dec. 2011.

R. Velazco, W. Mansour, F. Pancher, G. Marques-Costa, D. Sohier, A. Bui, Improving SEU Fault Tolerance Capabilities of a Self-Converging Algorithm, IEEE Transactions on Nuclear Science, vol. 59, pp. 818-823, March 2012.

W. Mansour, R.Velazco, SEU Fault-Injection in VHDL-Based Processors: A Case Study, candidated for publication in JETTA special issue (Journal of Electronic Testing: Theory and Applications), 2012.

Conférences et Workshops

R. Velazco, G. Foucard, F. Pancher, W. Mansour, G. Marques-Costa, D. Sohier, A. Bui, Robustness with respect to SEUs of a self-converging algorithm, 12th Latin-American Test Workshop (LATW'11), Porto de Galinhas, Brazil, 27-30, March 2011.

W. Mansour, R. Velazco, An automated SEU fault-injection method and tool for HDL-based designs, IEEE proceedings of RADECS 2012, Biarritz, France, 24-28 September 2012.

G. Hubert, R. Velazco, C. A. Federico, A. Cheminet, C. Silva-Cardenas, L.V.E. Caldas, F. Pancher, V. Lacoste, F. Palumbo, W. Mansour, L. Artola, F. Pineda, S. Duzellier, Continuous high-altitude measurements of cosmic ray neutrons and SEU/MCU at various locations: correlation and analyses based-on MUSCA SEP^3, IEEE proceedings of RADECS 2012, Biarritz, France, 24-28 September 2012.

W. Mansour, R. Ayoubi, H. Ziade, W. El Falou, R. Velazco, A Fault-Tolerant Implementation on FPGA of a Hopfield Neural Network, accepted for presentation at DCIS (Design of Integrated Circuits and Systems), Avigon, France, 28-30 November.

W. Mansour, W. El Falou, H. Ziade, R. Ayoubi, R. Velazco, SEU Simulation by Fault Injection in PSOC Device: Preliminary Results, 2nd International Conference on Advances in Computational Tools for Engineering Applications, ACTEA 12, Zouk-mosbeh, Lebanon.

C. A. Federico, R. Velazco, F. Pancher, C. Silva-Calderas, F. Pineda, G. Hubert, F. Palumbo, O.L. Goncalez, W. Mansour, S. Duzellier, L. V. Caldas, Medições de nêutrons oriundos de radiação cósmica em Puno (Peru) – uma participação no projeto HARMLESS, Atividades de Pesquisa e Desenvolvimento, IEAv, Vol. 5, P. 111, 2012.

Bibliographie

[Aguirre 2005] M.A. Aguirre et al., “FT-UNSHADES: A new 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.

[Aguirre 2007] M.A. Aguirre et al., “Selective Protection Analysis Using a SEU Emulator: Testing Protocol and Case Study Over the Leon2 Processor”, IEEE Trans. On Nucl. Sci., pp. 951-956, Vol. 54, NO.4, August 2007.

[Alderighi 2010] M. Alderighi al., “Experimental validation of fault injection analyses by the FLIPPER tool”, Trans. on the nuclear science, vol 57, no. 4, pp. 2129-2134, Aug. 2010.

[Antoni 2001] L. Antoni et al., “Using run-time reconfiguration for fault injection applications”, in Proceedings of the IEEE Instrumentation and Measurement Technology Conference, pages, 1773-1777, Budapest, May 2001.

[Arlat 1990] J. Arlat et al., “Fault-injection for dependability validation – a methodology and some applications”, Trans. on the IEEE Software Engineering, pp. 166-182, Feb. 1990.

[Artola 2011] L. Artola et al., “SEU prediction from SET modeling using multi-node collection in bulk transistors and SRAMs down to the 65 nm technology node,” IEEE Transactions on Nuclear Science, vol. 58, no. 3, Jun. 2011.

[Ayoubi 2002] R. A. Ayoubi et al., “An efficient implementation of multilayer perceptron on mesh architecture,” in Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS ’02), vol. 2, pp. 109–112, May 2002.

[Ayoubi 2004] R. A. Ayoub et al., “Hopfield associative memory on mesh” in Proceedings of the IEEE International Symposium on Circuits and Systems, pp. 800–803, May 2004.

[Baraza 2005] J.C. Baraza et al., “Improvement of fault injection techniques based on VHDL code modification” in Proceedings of 10th IEEE international High-Level Design Validation and Test Workshop, pages 19-26, 2005.

[Baumann 1995] R. Baumann et al., Boron compounds as a dominant source of alpha particles in semiconductor devices. IEEE Int. Reliability Physics Symposium, pages 297–302, 1995.

[Binder 1975] D.Binder et al. Satellite anomalies from galactic cosmic rays. IEEE Trans. Nucl. Sci., 22(6): 2675, 1975.

[Bou 1998] J. Bou et al., “MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerant”, in Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 168–173, June 1998.

[Boudenot 1995] J.C Boudenot, ‘’L’Environnement Spatial’’ Collection ‘’Que sais-je ?’’ Ed. Presses Universitaires de France, 1995.

[Boudenot E3950] J. Boudenot. Tenue des circuits aux radiations ionisantes, volume E 3 950. Techniques de l’Ingenieur.

[Bourrieau 1991] J.Bourrieau. L’Environnement Spatial (flux, dose, blindage, effets des ions lourds). Septembre 1991.

[Brenni 1999] P. Brenni. Le générateur de Van de Graaf, Une machine électrostatique pour le XXème siècle, Bulletin of the Scientifique Instrument Society No. 63, 1999.

[Brien 1971] K.O’Brien, “The Natural Radiation Environment” Report N°720805-P1, United States Departement of Energy, 1971, p.15.

[Brien 1978] K.O’Brien, Report N°EML-338, United States Departement of Energy, 1978.

[Buchner 1987] S. Buchner et al. Laser simulation of single event upsets. IEEE Trans. Nucl. Sci., 34(6): 1228, 1987.

[Chau 2000] S .N. Chau, L. Alkalai, A. T. Tai, “Analysis of a multi-layer fault-tolerant COTS architecture for deep space missions”, Application-Specific Systems and Software Engineering Technology, 2000.

[Chee 2000] A. Chee et al., ‘’Potential doses to passengers and crew of supersonic transports’’, Health Phys., Vol. 79, pp. 547, 2000.

[Cheynet 1999] P. Cheynet. Etude de la robustesse du contrôle intelligent face aux fautes induites par les radiations, PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[Civera 2001] P. Civera et al., ‘’ Exploiting FPGA-based techniques for fault-injection campaigns on VLSI circuits’’ IEEE International Symposium on Defect and Fault Tolerance in VLSI systems pp. 250-258, 2001.

[Delarochette 1995] H. De La Rochette, « Latchup déclenché par ion lourd dans des structures CMOS-1 µm : approche expérimentale, simulation 2D et 3D », Thèse, Université Montpellier II, 1995.

[Delong 1996] T.A Delong et. al., “A fault injection technique for VHDL behavioral-level models”, Proc. of the IEEE Design and Test of Computers, pp. 24-33, 1996.

[Dijikstra 1974] E.W. Dijikstra, “Self stabilizing systems in spite of distributed control”, Communications of the Association of the Computing Machinery, 17(11):643-644, 1974.

[DiUbaldo 2000] J. A. DiUbaldo, "NASA Earth Observing System Mission Operations Center development using COTS products", Aerospace Conference Proceedings, 2000.

[Doley 1995] Shlomo Doley, Amos Israeli and Shlomo Moran, "Analyzing Expected Time by Scheduler-Luck Games", IEEE Trans. Soft. Eng, 21(5): 429-439, 1995

[Doley 2000] Shlomy Doley, Self-Stabilization, MIT Press, 2000.

[Duzellier 1997] S. Duzellier, et al, "EXEQ II and III: On-board experiments for the study of single events", 4th Radiation and Its Effects on Components and Systems (RADECS'97), Cannes, France, pp. 504-509, 1997.

[Duzellier 2004] S. Duzellier. Space Radiation Environment and its Effects on Spacecraft Components and Systems, chapter Single Event Effects: analysis and testing, pages 221–242. 2004.

[Duzellier 2006] S.Duzellier. Space Radiation Environment and its Effects on Spacecraft Components and Systems, chapter Single Event Effects: analysis and testing, pages 221-242. 2004.

[Eddington 1926] Eddington, A. S., The Internal Constitution of Stars (Cambridge: Cambridge Univ. Press 1926.

[Eddy 2009] John A. Eddy, The Sun, The Earth and Near-Earth Space, National Aeronautics and Space Administration (NASA), NASA Publication #NP-2009-1-066-GSFC

[Entrena 2012] L. Entrena et al., Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection, Trans. On Computers, pp. 313-322, 2012

[Faure 2002] F. Faure et al.. Thesic+: A flexible system for see testing. In proc. Of RADECS, 2002.

[Federico 2012] C. Federico et al. Medições de nêutrons oriundos de radiação cósmica em Puno (Peru) – uma participação no projeto HARMLESS, Atividades de Pesquisa e Desenvolvimento, IEAv, Vol. 5, P. 111, 2012.

[Fleischer 1975] R. Fleischer et al. Nuclear tracks in solids, principles & applications. University of California press, 1975.

[Folkesson 1998] P. Folkesson et al., "A comparison of simulation based and scan chain implemented fault injection", Proc. Of the Annual International Symposium on Fault-Tolerant Computing, Jun. 1998, pp. 284-293.

[Foucard 2008] G. Foucard. Taux d'erreurs dues aux radiations pour des applications implémentées dans des FPGAs a base de mémoire SRAM: prédiction versus mesures. PhD thesis, Institut National Polytechnique de Grenoble, 2010.

[Goldhagen 2000] P.Goldhagen, ''Overview of aircraft radiation exposure and recent ER-2 measurements'', Health Phys., vol. 79, p.526, 2000.

[Guenzer 1979] C. S. Guenzer, et al., “Single event upset of dynamic ram's by neutrons and protons" IEEE Transactions on Nuclear Sciences, Vol. 26, December 1979.

[Guthaus 2001] M.R. Guthaus et al., ‘Mibench: A free, commercially representative embedded benchmark suite’’, In Proc. Of IEEE International Workshop on Workload Characterization, WWC-4, pages 3-14,200.

[Hopfield 1982] J. J. Hopfield, “Neurons with graded response have collective computational properties like those of two-state neurons,” Proceedings of the National Academy of Sciences of the United States of America, vol. 79, 1982.

[Huang 2005] Tetz C. Huang, “A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity”, Journal of Computer and System Sciences, 71(1): 70-85, 2005.

[Hubert 2009] G. Hubert et al. “Operational SER calculations on the SAC-Corbit using themulti-scales single event phenomena predictive platform (MUSCA Sep3),” IEEE Transactions on Nuclear Science, vol. 56, no. 6, pp. 3032–3042, Dec. 2009.

[Hubert 2010] G. Hubert et al., “Impact of the solar flares on the SER dynamics on micro and nanometric technologies,” IEEE Trans. Nucl. Sci.,vol. 57, no. 6, pp. 3127–3134, Dec. 2010.

[Jeitler 2009] M. Jeitler, “FuSE – a hardware accelerated HDL fault injection tool”, In SPL. 5th Southern Conference on Programmable Logic, 2009, pp. 84-94, April 2009.

[Jenn 1994] E. Jenn et al., “Fault injection into VHDL Models: The MEFISTO TOOL”, Proc. of the International Symposium on Fault-Tolerant Computing, pp. 336–344, June 1994.

[JESD89 2001] JEDEC Standard, ‘’Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices’’ revision of JESD89, August 2001.

[Kilpatric 1995] D.Kilpatric et al., “Unsupervised classification of Antarctic Satellite Imagery using kohonen’s self-organizing feature map ”, in Proceedings of IEEE International Conference on Neural Networks, , pp. 32-36, Perth, Australia, 1995.

[Kwiatkowska 1995] E. Kwiatkowska et al., “Hybrid neural network system for cloud classification from satellite images”, in Proceedings of The IEEE International Conference on Neural Networks, , pp. 1907-1912, Perth, Australia, 1995.

[Leiner 2008] B. J. Leiner et al., “Hardware architecture for FPGA implementation of a neural network and its application in images processing,” in Proceedings of the 5th Meeting of the Electronics, Robotics and Automotive Mechanics Conference (CERMA ’08), pp. 405–410, October 2008.

[Lesea 2005] A. Lesea et al., The rosetta experiment: Atmospheric soft error rate testing in differing technology fpgas. IEEE Trans. Device Mater. Rel., 5(3), Sept. 2005.

[Leung 2000] W. Leung et al. "The ideal SoC memory: 1T-SRAM", in Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, pp. 32-36, 2000.

[Lopez-Ongil 2007] C. Lopez-Ongil et al., "A unified Environment for Fault Injection at Any Design Level Based Emulation", IEEE Trans. On Nucl. Sci., Vol 54, pp.946-950, 2007.

[Madeira 1994] T.C. May et al., "Alpha-particle-induced soft errors in dynamic memories", IEEE Trans. On Electronic Devices, Vol 26, February 1979.

[May 1979] H.Madeira et al., "RIFLE: a general purpose pin-level fault injector", Proc of the European Dependable Computing Conference., pp. 199-216, 1994.

[Miniere 1996] X.Miniere et al., "A neural approach to the classification of electron & proton whistlers", in Journal of The Atmospheric & Terresrial Physics., vol 58, no. 7 pp. 911-924, 1996.

[Miranda 2007] H. Guzman-Miranda et al., "FT-UNSHADES-up: A platform for the analysis and optimal hardening of embedded systems in radiation environments", IEEE International Symposium on Industrial Electronics, pp. 2276-2281, 2008.

[Mohammadi 2012] A. Mohammadi et al., "SCFIT: A FPGA-based fault injection technique for SEU fault model", in proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 586-589, 2012.

[Naviner 2011] L. Naviner et al., "FIFA: A fault-injection-fault-analysis-based tool for reliability assessement at RTL level" Microelectronics Reliability, vol. 51, pp. 1459-1463, 2011.

[Nicolaidis 2010] M. Nicolaidis, "Soft Errors in Modern Electronic Systems", chapter 5.

[Normand 1996] E. Normand. Single-event effects in avionics. IEEE Transaction on Nuclear Science, 43(15): 461-474, April 1996.

[Oldham 2003] T. R. Oldham and F. B. McLean. Total ionizing dose effects in mos oxides and devices. IEEE Transaction on Nuclear Science, Vol 50, pp 483–499, June 2003.

[Olsen 1993] J.Olsen et al., ''Neutron-induced single event upsets in static rams observed at 10 km flight altitude'', IEEE Transaction on Nuclear Science, Vol. 40, pp 74-77, 1993.

[Peronnard 2008] P.Peronnard et al., « Predicting the SEU Error Rate through Fault Injection for a Complex Microprocessor », 2008 IEEE International Symposium on Industrial Electronics (ISIE'08), Cambridge, UK, June 30th – July 2nd, 2008.

[Peronnard 2009] P. Peronnard. Méthodes et outils pour l'évaluation de la sensibilité de circuits intégrés avancés face aux radiations naturelles, PhD thesis, Institut National Polytechnique de Grenoble, 2009.

[Peronnard1 2009] P. Peronnard et al., "Real-life SEU experiments on 90 nm SRAMS in atmospheric environment: measures versus predictions done by means of MUSCA SEP3 platform," IEEE Transactions on Nuclear Science, vol. 6, no. 6, pp. 3450–3455, Dec. 2009.

[Pouget 2001] V.Pouget et al., "Theorical Investigation on an Equivalent Laser LET", Microelectronics Reliability, Vol. 41, pp. 1513-1518, 2001.

[Rahbaran 2004] B. Rahbaran et al., "Built-in fault injection in hardware- the FIDYCO example", Proc. of the Second IEEE International Workshop on Electronic Design, Test and Application, pp. 327-332, January 2004.

[Rezgui 2001] S. Rezgui. Prédiction du taux d'erreurs d'architectures digitales: une méthode et des résultats expérimentaux, PhD thesis, Institut National Polytechnique de Grenoble, 2001.

[Richter 1987] A. Richter et al. Simulation of heavy charged particle track using focuses laser beams. IEEE Trans. Nucl. Sci., 34(6) :1234, 1987.

[Ritter 1999] J.C. Ritter, "Microelectronis and Photonics Test Bed", 20th Annual ASS Guidance and Control Conference, Breckenridge, Colorado, USA, February 5-9th, 1997.

[Rousselle 2001] Rousselle C. et al., "A Register-Transfer-Level Fault Simulator for Permanent and Transient Faults in Embedded Processors," in Proceedings of DATE'2001 Conference, Munich, Germany, 2001.

[Rumelhart 1986] Rumelhart D.E, McClelland, J.L. && PDP Research Group, Parallel Distributed Processing Vol.1: Foundations, Cambridge: MIT Press, 1986.

[Saif 2006] S. Saif et al., "An FPGA implementation of a hopfield optimized block truncation coding," in Proceedings of the 6th International Workshop on System on Chip for Real Time Applications (IWSOC '06), pp. 169–172, December 2006.

[Saif 2007] S. Saif et al., "An FPGA implementation of a neural optimization of block truncation coding for image/video compression," Microprocessors and Microsystems, vol. 31, no. 8, pp. 477–486, 2007.

[Shokrallah 2008] M. Shokrallah-Shirazi et al., "FPGA-based fault injection into synthesizable verilog HDL models", The Second International Conference on Secure System Integration and Reliability Improvement, SSIRI 2008.

[Sieh 1997] V. Sieh et al. "VERIFY: evaluation of reliability using VHDL-models with embedded fault description" Proc. of the International Symposium on Fault-Tolerant Computing, pp. 32–36, June 1997.

[Sierawki 2009] B.D. Sierawki et al. , "Impact of low-energy proton induced upset on test methods and rate predictions," IEEE Transactions on Nuclear Science, vol. 46, no. 6, pp. 3085–3092, Dec. 2009.

[Sohn 2000] J.H. Sohn, ‘’The effects of single event upsets in avionics systems’’, M.S. thesis, lowa State Univ., May 2000.

[Stapor 1988] W.J. Stapor et al., ‘’Charge Collection in Silicon for Ions of Different Energy but Same Linear Energy Transfer (LET)’’, IEEE Trans. Nuc. Sci., vol. 35, No 6 p. 1585, 1988.

[Stepanova 2007] M. Stepanova et al., “A hopfield neural classifier and its FPGA implementation for identification of symmetrically structured DNA motifs,” Journal of VLSI Signal Processing, vol. 48, no. 3, pp. 239–254, 2007.

[Taber 1992] A.Taber et al., ‘’Investigation and characterization of SEU effects and hardening strategies in avionics‘’, IBM Rep. 92-L75-020-2

[Taber 1993] A.Taber et al., ‘’Single event upset in avionics’’, IEEE Trans. on Nuc. Sci, vol NS-40, no. 2, pp. 120-126

[Tang 2004] H.K. Tang et al. Semm2: A modelling system for single event analysis. IEEE Transanctions on Nuclear Science, 51(6), 2004

[Velazco 1998] R. Velazco, et al., Thesic: A testbed suitable for the qualification of integrated circuits devoted to operate in harsh environment. In IEEE European Test Workshop, pages 89-90, 1998.

[Velazco 2000] R. Velazco et al., ‘’Predicting Error Rate for Microprocessor-Based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection’’, IEEE Transaction of Nuclear Science, Vol. 47, pp. 2405-2411, 2000.

[Waldemark 1995] J.Waldemark et al., “Hybrid neural network pattern recognition system for satellite measurements”, in Proceedings of The IEEE International Conference on Neural Networks, pp. 195-199, Perth, Australia, 1995.

[Web cisco] Cisco Systems, "Increasing Network Avalability" http://www.cisco.com/warp/public/779/largeent/learn/technologies/ina/IncreasingNetworkAvailability-WhitePaper.pdf

[Web cygwin] http://www.cygwin.com

[Web cypress] http://www.cypress.com

[Web gaisler] http://www.gaisler.com

[Web geoshaft] http://geoshaft.space.qinetic.com/qarm

[Web icyflex] http://www.csem.ch/docs/Show.aspx/11926/docname/A4-icyflex.pdf

[Web MPTB] http://www.nrl.navy.mil/content.php?P=04REVIEW209

[Web NRL] http://www.nrl.navy.mil/

[Web opencores] http://www.opencores.org

[Web tuwien] https://trac.ecs.tuwien.ac.at/Spear2/wiki/spear2_core

[Web vlsicad] http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.gaisler.com/products/leon2/leon.html

[Web STRV] http://www.lasp.colorado.edu/strv/index.shtml

[Weller 2009] R. A.Weller et al "General framework for single event effects rate prediction in microelectronics," IEEE Transactions on Nuclear Science, vol. 56, no. 6, pp. 3098–3108, Dec. 2009.

[Zarandi1 2003] H.R. Zarandi et al, "Fault injection into verilog models for dependability evaluation of digital systems", Proc. of the International Symposium on Parallel and Distributed Computing., pp. 281-287, October 2003.

[Zarandi2 2003] H.R. Zarandi et al., "Dependability analysis using a fault injection tool based on synthesizability of HDL models", Proc. of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, PP. 485-492, November 2003.

[Zheng 2008] H. Zhang et al., "FITVS: A FPGA-based Emulation Tool For High Efficiency Hardness Evaluation", in International Symposium on Parallel and Distributed Processing with Applications, pp. 525-531, 2008.

[Zeigler 1996] J.F. Ziegler et al., '' IBM experiments in soft fails in computer electronics (1978-1994),'' IBM Journal of Research and Development vol. 40, no. 1, pp. 3-18, January 1996.

TITRE

Méthodes et outils pour l'analyse tôt dans le flot de conception de la sensibilité aux soft-erreurs des applications et des circuits intégrés

RESUME

La miniaturisation des gravures des transistors résulte en une augmentation de la sensibilité aux soft-erreurs des circuits intégrés face aux particules énergétiques présentes dans l'environnement dans lequel ils opèrent. Une expérimentation, présentée au cours de cette thèse, concernant l'étude de la sensibilité face aux soft-erreurs, dans l'environnement réel, des mémoires SRAM provenant de deux générations de technologies successives, a mis en évidence la criticité de cette thématique. Cela pour montrer la nécessité de l'évaluation des circuits faces aux effets des radiations, surtout les circuits commerciaux qui sont de plus en plus utilisés dans les applications spatiales et avioniques et même dans les hautes altitudes, afin de trouver les méthodologies permettant leurs durcissements. Plusieurs méthodes d'injection de fautes, ayant pour but l'évaluation de la sensibilité des circuits intégrés face aux soft-erreurs, ont été le sujet de plusieurs recherches.

Les travaux réalisés au cours de cette thèse ont eu pour but le développement d'une méthode automatisable, avec son outil, permettant l'émulation des effets des radiations sur des circuits dont on dispose de leurs codes HDL. Cette méthode, appelée NETFI (NETlist Fault Injection), est basée sur la manipulation de la netlist du circuit synthétisé pour permettre l'injection de fautes de types SEU, SET et Stuck_at. NETFI a été appliquée sur plusieurs architectures pour étudier ses potentialités ainsi que son efficacité. Une étude sur un algorithme tolérant aux fautes, dit *self-convergent*, exécuté par un processeur LEON3, a été aussi présenté dans le but d'effectuer une comparaison des résultats issus de NETFI avec ceux issus d'une méthode de l'état de l'art appelée CEU (Code Emulated Upset).

MOTS CLEFS

Environnement spatial, événements singuliers, mémoires SRAM, environnement atmosphérique, injection de fautes, tolérance aux fautes, code HDL

TITLE

Methods and tools for the early analysis in the design flow of the sensitivity to soft-errors of applications and integrated circuits

ABSTRACT

Reducing the dimensions of transistors increases the soft-errors sensitivity of integrated circuits to energetic particles present in the environments in which they operate. An experiment, presented in this thesis, aiming to study soft-errors sensitivity, in real environment, of SRAM memories issued from two successive technologies, put in evidence the criticality of this thematic. This is to show the need to evaluate circuit's sensitivity to radiation effects, especially commercial circuits that are used more and more for space and avionic applications and even at high altitudes, in order to find the appropriate hardening methodologies. Several fault-injection methods, aiming at evaluating the sensitivity to soft-errors of integrated circuits, were goals for many researches.

In this thesis was developed an automated method, and its corresponding tool, allowing the emulation of radiation effects on HDL-based circuits. This method, so-called NETFI (NETlist Fault-Injection), is based on modifying the netlist of the synthesized circuit to allow injecting faults of different types (SEU, SET and Stuck_at). NETFI was applied on different architectures in order to assess its efficiency and put in evidence its capabilities. A study on a fault-tolerant algorithm, so-called *self-convergent*, executed by a LEON3 processor, was also presented in order to perform an objective comparison between the results issued from NETFI and those issued from another state-of-the-art method, called CEU (Code Emulated Upset).

.

KEYWORDS

Space environment, single event effects, SRAM memories, atmospheric environment, fault-injection, fault-tolerant algorithms, fault tolerance, HDL code

INTITULE ET ADRESSE DU LABORATOIRE

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

ISBN : 978-2-84813-193-1