Fault injection using run-time reconfiguration of FPGAs

L. Antoni

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

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

présentée et soutenue publiquement

par

Lorinc ANTONI

le 19 septembre 2003

Titre :

Injection de Fautes par Reconfiguration Dynamique de Réseaux Programmables

_____

Directeur de thèse : Régis LEVEUGLE
Co-directeur de thèse : Béla FEHER

_____

JURY

M.	Pierre GENTIL	, Président
Mme	Marta RENCZ	, Rapporteur
M.	Serge PRAVOSSOUDOVITCH	, Rapporteur
M.	Régis LEVEUGLE	, Directeur de thèse
M.	Béla FEHER	, Co-directeur de thèse

ECOLES DOCTORALES ET LEURS SPECIALITES DE THESES

- **Spécialités préparées à l'INPG** Spécialités préparées dans une autre université

ECOLE DOCTORALE « ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE, TELECOMMUNICATIONS, SIGNAL » (EEATS) **INPG**/UJF

- **AUTOMATIQUE, PRODUCTIQUE**
- **GENIE ELECTRIQUE**
- **MICROELECTRONIQUE**
- **OPTIQUE, OPTOELECTRONIQUE ET MICROONDES**
- **SIGNAL, IMAGE, PAROLE, TELECOMS**

ECOLE DOCTORALE « MATERIAUX ET GENIE DES PROCEDES » (MGP) **INPG**/UJF

- **ELECTROCHIMIE**
- **GENIE DES PROCEDES**
- **SCIENCE ET GENIE DES MATERIAUX**

ECOLE DOCTORALE « MECANIQUE ET ENERGETIQUE » **INPG**/UJF

- **ENERGETIQUE PHYSIQUE**
- **MECANIQUE DES FLUIDES ET TRANSFERTS**
- **MECANIQUE : CONCEPTION, GEOMECANIQUE, MATERIAUX**

ECOLE DOCTORALE « ORGANISATION INDUSTRIELLE ET SYSTEMES DE PRODUCTION » **INPG**/UPMF

ECONOMIE INDUSTRIELLE

- **GENIE INDUSTRIEL**

SOCIOLOGIE INDUSTRIELLE

ECOLE DOCTORALE « INGENIERIE POUR LE VIVANT : SANTE, COGNITION, ENVIRONNEMENT » **UJF**/INPG

GENIE BIOLOGIQUE ET MEDICAL

METHODES DE RECHERCHE SUR L'ENVIRONNEMENT ET LA SANTE

- **MODELES ET INSTRUMENTS EN MEDECINE ET BIOLOGIE**
- **SCIENCES COGNITIVES**

SPORT ET PERFORMANCE, FACTEURS BIOMECANIQUES ET ENVIRONNEMENT SOCIAL

ECOLE DOCTORALE « MATHEMATIQUES, SCIENCES ET TECHNOLOGIE DE L'INFORMATION » **UJF**/INPG

ENVIRONNEMENTS INFORMATIQUES POUR L'APPRENTISSAGE HUMAIN ET DIDACTIQUE

- **IMAGERIE, VISION ET ROBOTIQUE**
- **INFORMATIQUE : SYSTEMES ET COMMUNICATIONS**

MATHEMATIQUES

- **MATHEMATIQUES APPLIQUEES**
- **RECHERCHE OPERATIONNELLE, COMBINATOIRE ET OPTIMISATION**
- **SYSTEMES D'INFORMATION**

ECOLE DOCTORALE « PHYSIQUE » **UJF**/INPG

ASTROPHYSIQUE ET MILIEUX DILUES

CRISTALLOGRAPHIE ET RMN BIOLOGIQUES

- **METHODES PHYSIQUES EXPERIMENTALES ET INSTRUMENTATION**
- **PHYSIQUE DE LA MATIERE ET DU RAYONNEMENT**
- **PHYSIQUE DES MATERIAUX : DES NANOSTRUCTURES AUX GRANDS INSTRUMENTS**

ECOLE DOCTORALE « TERRE, UNIVERS, ENVIRONNEMENT » **UJF**/INPG

ASTROPHYSIQUE ET MILIEUX DILUES

CLIMAT ET PHYSICO-CHIMIE DE L'ATMOSPHERE

DYNAMIQUE DE LA LITHOSPHERE

- **MECANIQUE DES MILIEUX GEOPHYSIQUES ET ENVIRONNEMENT**

POLLUTION ET ENVIRONNEMENT : PHYSICO-CHIMIE, IMPACT ET MODELISATION

Table of Contents

Abstract

Fault injection techniques have been used for a long time to evaluate the dependability of a given hardware, software or system implementation. The basic idea is to deliberately create faults into the environment under test after putting it into operation. The system under test is excited with application test vectors and data are collected on the outputs and also potentially on internal signals. At the end, these data can be used in order to analyse the behaviour of the system when faults occur.

This work focuses on hardware-based fault injections, in digital circuits. In this context, it was proposed to take advantage of hardware prototyping to improve and accelerate the execution of the whole fault injection campaign. Reconfigurable hardware (and especially FPGA devices) is a good candidate to implement the prototypes used to perform fault injections. The reconfiguration of an FPGA can however take a long time and this can be a limitation of prototyping-based techniques, especially if the device must be reconfigured several times. To overcome this problem, the work presented in this thesis aims at taking advantage of partial (also called local) reconfiguration capabilities of the hardware. In this case, only a part of the device must be reconfigured when changes are made. The use of partial reconfiguration capabilities results in an important time gain when only few differences exist between two successive configurations of the FPGA.

Until now, hardware prototyping was used for the execution of the application on faulty versions of the circuit. The fault injection itself was generally made by means of internal logic elements controlled by external signals. These elements were added by modifying either the high level description (e.g. behavioural VHDL) or the gate level description of the circuit, before implementing the prototype. The idea developed in this thesis is not only to execute the application in reconfigurable hardware but also to realise the injection of faults directly in the device (FPGA), taking advantage of the reconfiguration capabilities. Like this, each fault injection (or fault removal) necessitates a partial reconfiguration of the device. On the other hand, the initial

description of the system must not be changed before implementing the prototype.

This thesis demonstrates the feasibility of such an approach, for two main types of faults (stuck-at faults and asynchronous bit-flips modeling Single Event Upsets). The injection process using partial reconfiguration has been automated for these types of faults in the case of Virtex-based prototypes. The advantages and limitations with respect to existing techniques have been analysed. Finally, the work concludes on the major parameters that must be optimised to implement an efficient fault injection system based on partial reconfiguration.

Résumé

Des techniques d'injection de fautes ont été utilisées depuis de nombreuses années pour évaluer la sûreté de systèmes ou de composants (matériels ou logiciels). Ces techniques sont fondées sur la création délibérée de fautes dans le système à tester, pendant l'exécution d'une application. Les sorties du système, et potentiellement certains signaux internes, sont enregistrés et ces données sont utilisées à la fin des expériences pour analyser le comportement du système en présence de fautes.

Le travail présenté dans cette thèse est focalisé sur des injections de fautes au niveau matériel, dans des circuits digitaux. Dans ce contexte, l'utilisation de prototypes a été proposé pour améliorer et accélérer la réalisation des campagnes d'injection. Les réseaux programmables (et en particulier les réseaux de type FPGA) sont de bons candidats pour implémenter de tels prototypes. La reconfiguration d'un FPGA peut toutefois nécessiter un temps assez long, ce qui peut constituer une limitation des techniques basées sur le prototypage, surtout si de nombreuses reconfigurations sont nécessaires pour réaliser l'injection des fautes. Afin de résoudre ce problème, cette thèse propose de mettre à profit les possibilités de reconfiguration partielle (ou locale) de certains réseaux. En utilisant cette possibilité, seule une partie du réseau doit être reconfigurée lorsque des modifications sont requises, ce qui conduit à des gains de temps notables lorsque seules quelques différences existent entre deux configurations successives.

Jusque là, le prototypage matériel n'a été employé que pour exécuter l'application sur des versions de circuits modifiées pour injecter les fautes souhaitées. L'injection elle-même était réalisée grâce à des dispositifs ajoutés dans le circuit et commandés par des signaux externes. Ces modifications étaient introduites soit dans la description de haut niveau (par exemple, VHDL comportemental) soit dans la description au niveau portes, avant d'implémenter le prototype. L'idée développée dans cette thèse est non seulement d'exécuter l'application sur un prototype, mais aussi de réaliser l'injection des fautes directement dans le composant (FPGA) en tirant profit des possibilités de

reconfiguration. De cette façon, chaque injection (ou suppression) de faute nécessite une reconfiguration partielle du FPGA. En revanche, la description initiale du circuit n'a pas à être modifiée avant l'implémentation du prototype.

Cette thèse démontre la faisabilité d'une telle approche, pour deux types de fautes majeurs (les collages et les inversions de bits asynchrones, qui modélisent les fautes de type *Single Event Upset*). Le processus d'injection utilisant la reconfiguration partielle a été automatisé pour ces types de fautes dans le cas de prototypes implémentés sur des réseaux Virtex. Les avantages et les limitations par rapport aux techniques existantes ont été analysés. Enfin, la thèse conclut sur les principaux paramètres devant être optimisés pour implémenter un environnement d'injection de fautes fondé sur la reconfiguration partielle.

Résumé étendu en Français

Introduction

Le but de la recherche effectuée dans le cadre de cette thèse de doctorat en cotutelle est d'exploiter les possibilités de l'application de la reconfiguration dynamique (partielle) dans le domaine de l'injection de fautes.

Les différentes techniques d'injection de fautes ont été reconnues comme nécessaires pour la validation de la sûreté d'un système en analysant le comportement des dispositifs quand une faute survient. Le matériel reconfigurable (réseaux programmables, tels que les FPGA) est approprié pour implémenter et tester des prototypes en synthétisant des descriptions de haut niveau (par exemple, en VHDL). De plus, le prototypage permet de faire des émulations "*in-system*" avant la fabrication des circuits, ce qui permet d'analyser les interactions temps réel du circuit avec l'environnement applicatif. L'application de l'injection de fautes dans un prototype matériel peut donc être une approche intéressante pour évaluer tôt dans le cycle de conception les caractéristiques de sûreté.

La reconfiguration dynamique (*Run-Time Reconfiguration* - RTR) est une technique bien connue qui correspond à la reconfiguration du matériel pendant l'exécution d'une application. Il existe plusieurs sortes de RTR, selon que le dispositif est reconfiguré totalement ou seulement partiellement à chaque pas de reconfiguration. Nous proposons d'utiliser la RTR, et plus particulièrement la RTR partielle, pour accélérer l'injection de fautes dans un prototype matériel.

Etat de l'art du calcul sur support reconfigurable

Les architectures reconfigurables ("*Reconfigurable computing*") évoquent la possibilité de pouvoir changer l'architecture d'un système informatique en temps-réel. Les ordinateurs de nos jours ont en général une structure matérielle figée, basée sur un

microprocesseur. Les composants figés peuvent être classifiés comme : logique (PLAs, réseaux de portes, etc), contrôle embarqué (e.g. contrôleurs ASICs, ou circuits VLSI) et microprocesseurs (e.g. x86, 68000, PowerPC). En même temps que la puissance des microprocesseurs augmente avec chaque nouvelle génération, le besoin de nouvelles fonctionnalités dans les applications entraîne une modification progressive des contraintes de conception des systèmes.

Les systèmes de *reconfigurable computing*, si on les compare au matériel figé, ont comme principale différence la possibilité de pouvoir changer leur architecture de manière logicielle pour les rendre optimaux pour l'exécution des différentes phases de l'application. On peut établir trois catégories de matériel reconfigurable :

- logique (FPGAs)
- contrôle embarqué (e.g. co-processeurs reconfigurables)
- ordinateurs (des plates-formes complètement reconfigurables utilisant des FPGAs dans un système qui a pour cible les applications générales)

Les circuits reconfigurables profitent des avantages du parallélisme, et ils peuvent avoir un surcoût moins important pour les accès mémoire, pour les opérations de branchement et pour le décodage des instructions. Les approches d'implémentation les plus importantes [64] du calcul sur support reconfigurable sont la reconfiguration statique (*Compile-Time Reconfiguration*, CTR) et la reconfiguration dynamique (*Run-Time Reconfiguration*, RTR). La reconfiguration statique est une stratégie statique où chaque application est constituée d'une seule configuration. La reconfiguration dynamique est une stratégie dynamique où une application est constituée de plusieurs configurations qui résident à des moments différents sur le support reconfigurable et qui collaborent à l'exécution de l'application globale. Dans les sections qui suivent, nous allons traiter ces deux classes de reconfiguration.

Reconfiguration statique (CTR)

La CTR est l'approche la plus simple et la plus utilisée pour l'implémentation des applications sur des circuits reconfigurables [64]. La caractéristique la plus importante des applications CTR est le fait qu'elles ne se sont constituées que d'une seule configuration matérielle. Avant l'exécution de l'opération, les FPGAs sont chargés avec leurs configurations, et une fois l'exécution de l'application commencée, ils restent dans cette configuration jusqu'à la fin de l'exécution (Fig. 1). Cette approche est

très similaire à l'utilisation des ASICs pour accélérer l'exécution d'une application. En effet, du point de vue de l'application cela revient dans ce cas au même d'utiliser des FPGAs ou des ASICs comme matériel car le matériel ne change pas au cours de l'exécution de l'application. La plus grande similarité entre la conception matérielle conventionnelle et la conception CTR est la stratégie statique d'allocation du matériel : dans les deux cas, le matériel est statique pour toute l'application. Bien que la technologie FPGA puisse permettre une approche dynamique, les applications CTR gardent la caractéristique statique pour simplifier le concept.

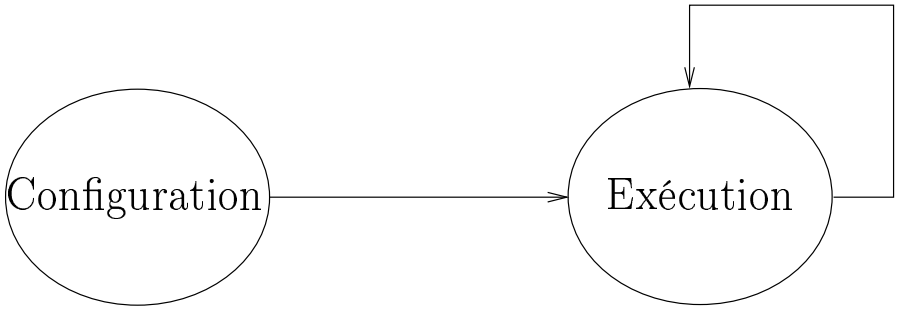


FIG. 1: Reconfiguartion statique (*CTR*)

Reconfiguration dynamique (RTR)

Alors que les applications CTR associent les éléments logiques d'une manière statique pour toute la durée de l'application, les applications RTR utilisent une allocation dynamique qui réorganise le matériel pendant l'exécution [95]. Chaque application est constituée de plusieurs configurations pour chaque FPGA, chaque configuration implémentant une fraction de l'application. Alors, contrairement aux applications CTR qui configurent les FPGAs définitivement une fois avant l'exécution, les applications RTR les reconfigurent plusieurs fois durant l'exécution d'une application donnée.

La nature dynamique du matériel introduit deux nouveaux problèmes [64]. Le premier est la partition de l'algorithme en segments qui ne doivent pas (ou qui ne peuvent pas) être exécutés concurremment (on appelle cela partitionnement temporel). Les éléments de chacun de ces segments seront exécutés concurremment, et on charge ces segments dans le matériel reconfigurable l'un après l'autre. Comme cela, les différents segments sont exécutés séquentiellement. Il est possible de réaliser un partitionnement temporel d'un algorithme en tenant compte des différentes phases de l'opération. Chaque phase peut être associée ensuite à un segment différent, c'est-à-dire à une configuration de FPGA. Ces configurations sont chargées ensuite dans les

FPGAs.

La reconfiguration dynamique a été introduite à cause de contraintes de ressources. Il n'est pas toujours possible d'implanter d'une manière statique des applications de taille importante car les ressources matérielles utilisables sont en général limitées. La reconfiguration dynamique permet d'utiliser moins de ressources car c'est toujours juste une partie de l'application qui réside dans le matériel, pas l'application dans sa globalité. Les configurations d'une application RTR doivent rester dans le circuit pour une période déterminée, et être remplaçables après cette période. Cela pose un problème car presque tous les outils de développement considèrent un modèle statique et donc ne tiennent pas compte de cette phase de remplacement lors du développement. La deuxième problématique introduite par les systèmes RTR est le contrôle du comportement entre les configurations. Ce comportement est constitué de la communication entre les différentes configurations et de l'échange des données entre elles : des configurations produisent des données intermédiaires qui formeront les données d'entrée de la prochaine configuration chargée. Les configurations doivent être ainsi construites d'une manière très rigoureuse en tenant compte des différentes phases de l'application pour que les données intermédiaires soient correctement gérées.

Deux approches de base existent pour implémenter des applications RTR : Globale ou Locale (celle-ci aussi appelée "partielle"). Les deux approches utilisent plusieurs configurations pour réaliser une seule application, et les deux reconfigurent les FPGAs pendant l'exécution de l'application. La différence entre les deux approches se trouve dans l'allocation du matériel. Nous allons examiner d'un peu plus près ces deux approches dans les deux sections suivantes.

RTR Globale

Pour la RTR Globale on implique toutes les ressources matérielles lors de chaque pas de reconfiguration. Comme cela il est possible de partager l'application en phases temporelles, et chaque phase représente une configuration qui occupe toutes les ressources reconfigurables (Fig. 2). Dans le cas d'une application utilisant la RTR Globale il faut diviser l'application en partitions plus ou moins de même taille pour avoir une utilisation efficace des ressources reconfigurables [64]. Il est bien évident que cette étape de division est itérative, car il est difficile de juger de la taille d'une partition avant le placement et le routage. Il faut créer une partition initiale et itérer ensuite tant sur le schéma que sur la partition jusqu'à ce qu'on obtienne des partitions de

tailles plus ou moins équivalentes. Une problématique élémentaire se pose car aucun outil ne peut simuler directement la transition d'une configuration à une autre. Mais le désavantage majeur de l'approche RTR Globale est la complexité de la tâche pour créer des partitions de même taille. Si on n'est pas capable de le faire, l'utilisation des ressources FPGA ne sera pas économique.

Une fois que l'on a partitionné notre application, il faut matérialiser la communication entre les différentes configurations. Puisque chaque phase occupe toutes les ressources reconfigurables, les interfaces entre les configurations peuvent être fixées et tous les modules peuvent être conçus dans le même contexte. Normalement, pendant l'exécution de l'application, chaque phase, après chargement de sa configuration, exécute sa tâche et place ensuite ses résultats intermédiaires dans une ressource fixée, par exemple dans une mémoire ou un registre. La configuration suivante va y lire ses données d'entrée. Ceci est répété jusqu'à la fin de l'application.

L'avantage le plus important de la RTR Globale est qu'elle est relativement simple. Une fois que le partitionnement est fait, des outils conventionnels peuvent être utilisés. Chaque partition peut être plus ou moins considérée comme un module indépendant et l'outil peut tenter de faire des optimisations sur la partition.

Plusieurs applications ont déjà été développées en RTR, par exemple le RRANN qui est un système appliquant la RTR aux réseaux de neurones [51] [52] [53]. A part cela, il existe de nombreuses autres applications utilisant la RTR et la reconfiguration partielle.

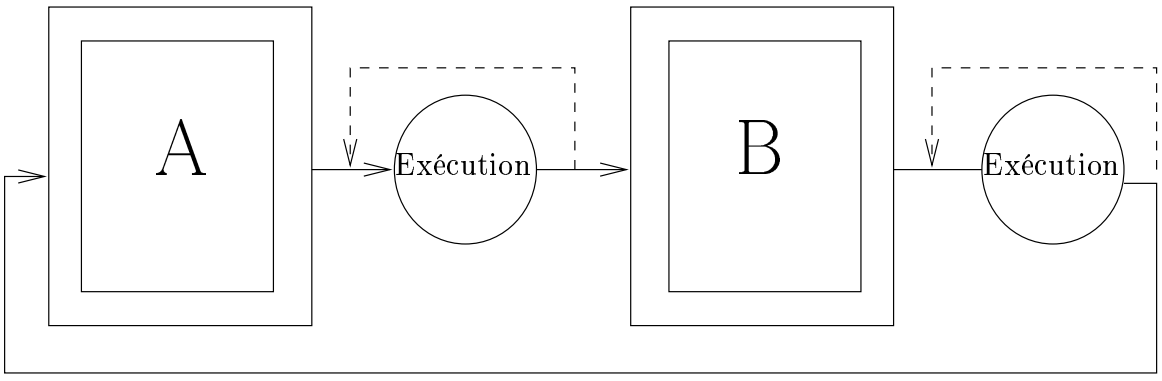


FIG. 2: RTR Globale

RTR Locale (partielle)

La RTR Locale (partielle) est une approche encore plus flexible que la RTR Globale. Comme on peut facilement le deviner, les applications en RTR partielle reconfigurent des sous-ensembles de logique d'une manière locale ou sélective pendant l'exécution (Fig. 3). N'importe quelles ressources peuvent être reconfigurées n'importe quand, même une partie d'un seul FPGA peut être reconfigurée. Cette flexibilité rend possible une allocation plus fine des ressources matérielles que celle permise par la RTR Globale. Contrairement à la RTR Globale, on charge seulement les éléments logiques nécessaires dans les ressources matérielles et on réduit ainsi le temps de chargement des configurations et on obtient une utilisation plus efficace du matériel. Un autre avantage de la RTR Locale est la possibilité de ne reconfigurer que les parties dynamiques du système, les fonctions statiques restant inchangées dans le FPGA tout au long de l'exécution.

La conception des applications avec RTR Locale est basée plutôt sur un partitionnement fonctionnel [64] contrairement au partitionnement en phases de la RTR Globale. Ces partitions ne doivent pas être entièrement exclusives puisque plusieurs d'entre elles peuvent être actives en même temps, alosr qu'avec la RTR Globale il n'y a qu'une seule partition qui est active à un instant donné. Toutes ces partitions ou opérations peuvent être considérées comme des modules indépendants qui sont chargés dans les FPGAs à la demande. Il est important de noter que plusieurs de ces partitions peuvent être chargées simultanément et que chaque partition peut occuper un nombre arbitraire de ressources des FPGAs.

Avec sa granularité fine, la RTR partielle a l'avantage principal sur la RTR Globale de mieux utiliser les ressources FPGA, tout particulièrement pour les applications qui ne peuvent pas être facilement divisées en partitions temporelles de taille identique. Malheureusement ce gros avantage représente un désavantage important aussi [64] puisque la complexité du système croît simultanément. Les interfaces dans le cas de la RTR Locale ne sont pas fixées, et peuvent changer avec chaque configuration. Quand plusieurs configurations sont chargées en même temps dans la même ressource il faut bien traiter leurs interfaces pour que la communication entre elles soit correcte. En plus, il faut s'occuper du placement et routage aussi pour assurer l'échange correct de données entre les configurations. Il faut aussi s'occuper du partitionnement matériel/logiciel concernant l'ordonnancement des processus [72].

Malheureusement les outils de conception ne supportent pas trop la RTR partielle.

Ils ne rendent possibles ni la simulation des transitions entre les configurations ni le contrôle du routage et du placement des configurations dans le contexte des autres configurations.

Un système qui applique la RTR Locale est RRANN-2 dans le domaine des réseaux de neurones et des processeurs programmables, ou bien le DISC (Dynamic Instruction Set Computer) [140].

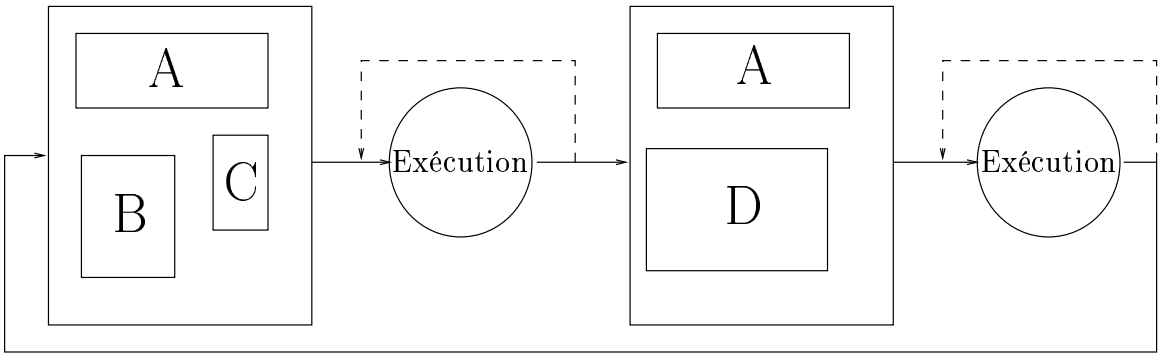


FIG. 3: RTR Locale

Injection de fautes : modèles VHDL et prototypes matériels

Comme mentionné précédemment, des techniques d'injection de fautes ont été proposées il y a déjà longtemps pour évaluer la sûreté d'un certain circuit ou d'une implémentation d'un système. La plupart des approches proposées jusqu'ici sont appliquées quand le système ou le circuit est disponible. Ces approches incluent l'injection de fautes sur les broches d'entrée/sortie, la corruption de mémoire [102], l'injection d'ions lourds, la perturbation de l'alimentation [82], l'injection de fautes par laser [120] ou l'injection de fautes logicielle [76].

Plus récemment, plusieurs auteurs ont proposé d'appliquer l'injection de fautes tôt dans le processus de conception. L'approche principale consiste à réaliser l'injection de fautes dans des modèles de haut niveau du circuit ou du système (le plus souvent, dans des modèles décrits en VHDL). [47] décrit par exemple l'injection de fautes dans des descriptions VHDL comportementales de systèmes à base de microprocesseur. [71] et [131] ou [23] proposent l'injection de différentes sortes de fautes dans le modèle VHDL d'un circuit à plusieurs niveaux d'abstraction et appliquent des techniques variées basées sur la description initiale VHDL ou des commandes de

simulation. Comme dans [47], des simulations sont utilisées pour évaluer l'impact des fautes sur le comportement du circuit. Comme mentionné dans [88], le désavantage principal de telles simulations est la grande quantité de temps nécessaire pour exécuter les expériences quand de nombreuses fautes doivent être injectées dans un circuit complexe.

Pour affronter les limitations en temps de simulation, il a été proposé de profiter du prototypage matériel, en utilisant un émulateur matériel ou un système reconfigurable à base de FPGA [87]. Un autre avantage de l'émulation est de permettre au concepteur d'étudier le comportement réel du circuit dans l'environnement de l'application, en considérant des interactions temps-réel [22]. Quand un émulateur est utilisé, la description VHDL initiale doit être synthétisable, bien sûr. Dans certains cas restreints, les approches développées pour l'émulation de fautes (e.g. [38], [139]) peuvent être appliquées pour injecter des fautes. Néanmoins, ces approches sont limitées en général à l'injection de fautes de collage (*stuck-at*). Dans le cas particulier de l'émulateur SimExpress, les fautes pouvaient aussi être injectées dans le circuit prototype en profitant des fonctionnalités embarquées de l'émulateur [1]. Toutefois, cette fonctionnalité n'existe pas dans les émulateurs actuellement disponible dans le commerce. de plus, cette sorte d'injection était limitée au modèle des fautes de collage simples. Dans la plupart des cas, des modifications doivent donc être introduites dans la description initiale pour pouvoir réaliser les injections de fautes. Il faut alors conserver une description synthétisable et satisfaire plusieurs contraintes liées au matériel d'émulation [88]. Les modifications peuvent donc être délicates à effectuer et il peut être nécessaire de générer plusieurs descriptions différentes, chacune permettant l'injection d'un certain ensemble de fautes. Dans ce cas l'émulateur matériel doit être reconfiguré complètement plusieurs fois, ce qui est une tâche qui nécessite beaucoup de temps (en tenant compte des temps de synthèse, placement et routage) et qui réduit donc le gain en temps d'exécution comparé à la simulation.

Le but principal de la recherche effectuée dans le cadre de cette thèse de doctorat était de proposer une nouvelle méthodologie permettant de profiter des avantages de l'émulation tout en évitant certains de ses inconvénients. Un avantage de la nouvelle méthodologie est d'éviter toute modification de la description initiale pour injecter une faute. Un autre avantage possible est la réduction du temps de reconfiguration global quand une campagne d'injection de fautes est exécutée sur un émulateur matériel (en évitant plusieurs configurations correspondant aux différentes descriptions modifiées).

Par ailleurs, le temps de reconfiguration est également réduit en tirant profit des possibilités de reconfiguration partielle.

Une nouvelle méthodologie utilisant le calcul sur support reconfigurable pour l'injection de fautes

Dans le cas du prototypage matériel "classique", l'injection de fautes est réalisée par modification du code VHDL. Dans un deuxième temps, le code est synthétisé et la description au niveau portes (*netlist*) est générée. Après placement et routage, la configuration du circuit peut être chargée dans le FPGA dans un format binaire (*bitstream*). Finalement, l'application est exécutée dans le matériel reconfigurable et une analyse du comportement en présence de fautes peut être réalisée. Pendant l'exécution des expériences, l'injection des fautes est commandée par des signaux externes ajoutés lors de la modification de la description du circuit.

Le flot de ce type d'injection de faute est montré en figure 4.

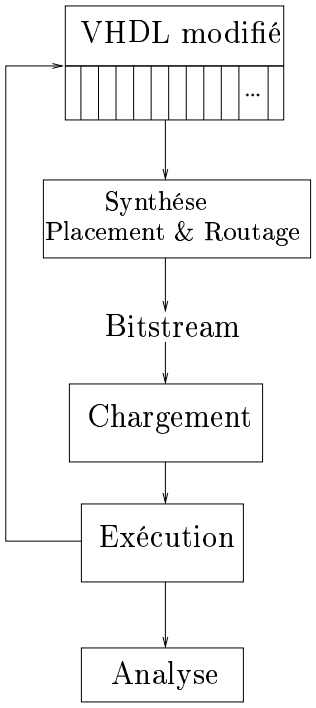


FIG. 4: Injection de faute en modifiant la description VHDL initiale

Dans cette approche, afin de tenir compte des limitations matérielles de l'émulateur (nombre de broches et nombre de portes), il peut être nécessaire de générer plusieurs versions modifiées du circuit contenant différentes sortes d'injecteurs de fautes

correspondant à des structures très similaires. Ces descriptions sont synthétisées et chargées séparément pour analyser le comportement du circuit avec les différentes fautes (Fig. 5).

La méthodologie développée dans le cadre de cette thèse est basée sur l'injection de fautes au plus bas niveau : la description initiale est synthétisée sans modification, et les fautes sont injectées directement dans le circuit pendant l'exécution sur l'émulateur en appliquant une approche RTR (partielle). Il est aussi possible d'injecter les fautes dans le fichier bitstream, et d'effectuer ensuite la reconfiguration partielle. Comme cela, le circuit ne doit pas être resynthétisé, mais seulement rechargé dans le matériel reconfigurable. Cette approche est montrée en figure 5.

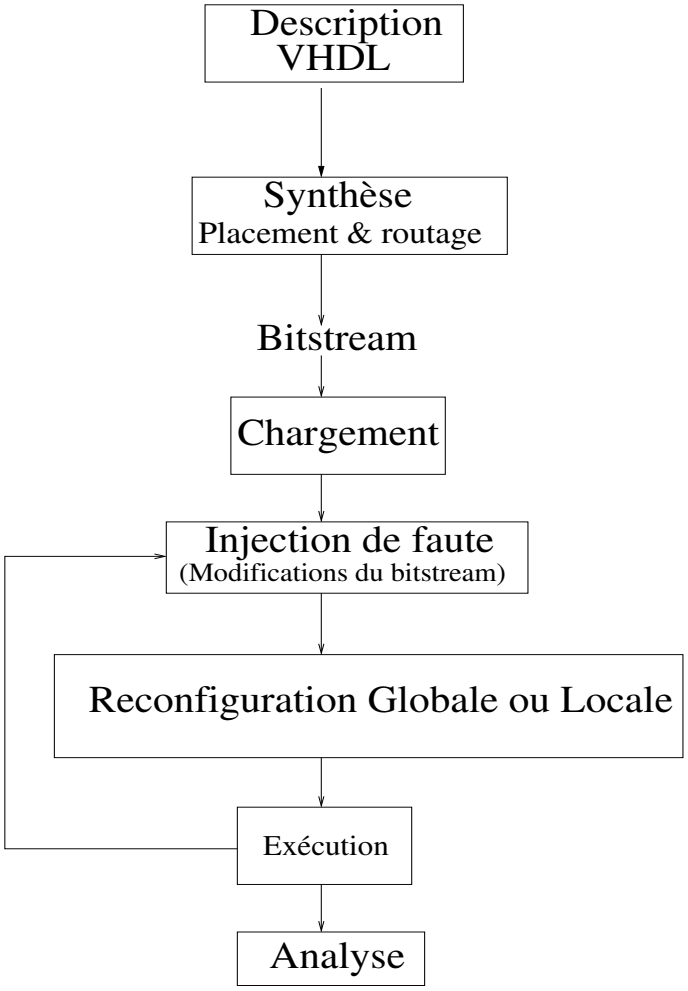


FIG. 5: La méthodologie proposée d'injection de fautes

La section suivante décrit les principes d'implémentation de la méthodologie proposée. Un aperçu de l'interface JBits, avec laquelle l'implémentation a été réalisé, est

donné ensuite.

L'implémentation de la méthodologie proposée

Avec la méthodologie proposée, l'injection de deux sortes de fautes a été réalisée : fautes de collage et *SEU*s. Les deux sections suivantes décrivent la technique d'injection de ces fautes dans le cas d'une implémentation sur réseau programmable Virtex (Xilinx).

L'injection de fautes de collage

Considérant la structure interne d'un réseau programmable Virtex, une fonction logique est placée dans un LUT (*Look-Up Table*) d'un Bloc Logique Configurable (CLB - *Configurable Logic Block*). Un LUT d'un FPGA Xilinx Virtex a 4 entrées (des signaux déduits de la description VHDL sont placés sur ces entrées), et la fonction du LUT est définie par 16 bits de mémoire.

Pour l'injection de fautes dans les blocs combinatoires, l'essentiel de l'approche consiste à changer les valeurs du LUT de façon à imiter l'effet de la faute (par exemple, le collage d'une des entrées). La figure 6 montre un exemple de réalisation d'une faute de collage. Le LUT est montré sous forme d'un tableau de Karnaugh.

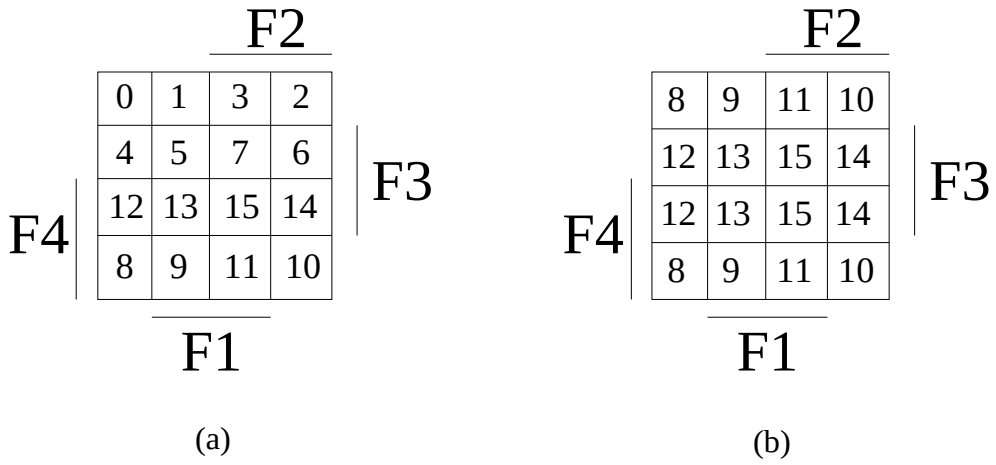


FIG. 6: Injection d'une faute de collage à 1 sur l'entrée F4 du CLB : (a) l'état initial du LUT (b) le LUT modifié

Cet exemple montre l'injection d'une faute de collage à 1 sur l'entrée F4. L'injection des fautes de collage à 0 et également de fautes d'inversion peut être réalisée de la même façon.

Injection des *SEU*s

L'implémentation d'un circuit dans un FPGA Virtex utilise les LUT d'un CLB pour les parties combinatoires. Les autres éléments importants des CLB sont les bascules (*flip-flop* en anglais, FF), employées pour implémenter les registres. Une faute de type *Single Event Upset* (*SEU*) correspond au basculement d'un tel élément mémoire, dont le contenu change de façon asynchrone de 1 à 0 ou de 0 à 1.

Dans la méthodologie proposée, des *SEU*s peuvent être injectés en changeant les états des bascules à l'inverse. Ce changement ne peut être fait que d'une seule manière (dans le cas du type de réseau considéré) : en appliquant un *set* ou un *reset* sur la bascule, en fonction de l'état au moment de l'injection.

Dans les FPGAs Virtex (utilisés pour les expériences), il y a une entrée globale GSR (*Global Set/Reset*) qui est connectée à l'entrée de tous les éléments principaux du FPGA (donc, aux FFs aussi). La fonctionalité du signal GSR peut prendre deux valeurs : *set* ou *reset*. La maniére la plus simple de décrire cela est de considérer un interrupteur qui détermine si un *set* ou un *reset* est appliqué à l'entrée de la bascule quand la ligne GSR est activée. La figure 7 illustre cela.

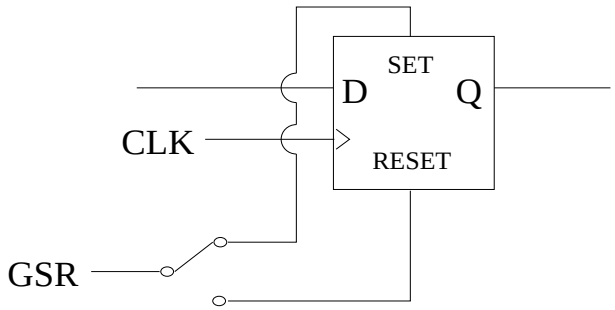


FIG. 7: Structure simplifiée d'une bascule dans le CLB d'un FPGA Virtex

Pour réaliser l'injection d'un SEU, il faut donc passer par les étapes suivantes :

- Lire l'état de toutes les bascules utilisées
- Comparer la position des interrupteurs avec les états des bascules
- Reconfigurer le FPGA pour changer l'état des interrupteurs de telle façon que le signal GSR ne modifie que le contenu de la bascule cible de l'injection
- Appliquer une impulsion sur la ligne GSR
- Reconfigurer le FPGA pour re-placer les interrupteurs dans leur position initiale

Comme on le voit, une lecture et deux reconfigurations doivent être faites pour injecter un *SEU*, ce qui peut prendre un temps important. Ce temps est réduit en appliquant une reconfiguration (et une lecture) partielle. Toutefois, la structure interne des FPGA Virtex limite l'effet de la reconfiguration partielle, toutes les bascules ou tous les interrupteurs similaires des CLB d'une même colonne devant obligatoirement être lus ou reconfigurés simultanément.

L'interface JBits

L' interface JBits est une API (Application Programming Interface) permettant de programmer des applications pour Virtex [59]. C'est un outil basé sur Java qui permet aux concepteurs d'écrire de l'information directement dans un FPGA Xilinx. JBits nous permet de modifier rapidement le bitstream du FPGA et d'effectuer une reconfiguration du circuit. Dans le cas des FPGAs Virtex, l'interface JBits permet aussi de reconfigurer partiellement la logique interne du dispositif. L'architecture Virtex maintient l'information temporaire pendant la phase de reconfiguration.

Les application JBits, ou "applets", peuvent aussi utiliser l'interface Java pour Boundary-Scan (sortie par Xilinx en Septembre 2000) pour effectuer des configurations localement ou à distance par Internet. Ces applets peuvent être des programmes de contrôle, des programmes d'interface utilisateur, ou des mises à jour. Auparavant, les applets Java étaient utilisés seulement pour envoyer des mises à jour logicielles par Internet. L'interface JBits permet maintenant de créer des applets Java de logique qui peuvent être utilisés pour envoyer des mises à jour matérielles par Internet.

Les principales caractéristiques de JBits

Le but de JBits est de fournir une interface rapide et simple vers les FPGAs des familles XC4000 et Virtex de Xilinx [144] (actuellement, seule la famille Virtex est supportée). La motivation initiale était de supporter des applications qui ont besoin d'une reconfiguration dynamique et rapide du circuit. En utilisant des outils traditionnels, la génération d'un circuit à partir d'une spécification VHDL ou d'un schéma peut prendre de plusieurs minutes jusqu'à plusieurs heures pour produire un nouveau bitstream. Mais même quelques secondes sont souvent trop pour les contraintes d'une application reconfigurable.

JBits a plusieurs caractéristiques qui le rendent unique et lui permettent d'adresser des problèmes qui ne sont pas traités par d'autres outils [144]. Les caractéristiques

les plus importantes sont :

- Compilation de 10.000 lignes par seconde (Microsoft J++ Java compiler)
- Support de reconfiguration dynamique et partielle
- Support pour paramétrer des circuits pendant l'exécution
- Support pour des macros pré-placées, pré-routées
- Modèle de programmation simple pour le contenu des CLBs
- Accès a toutes les ressources matérielles
- Support (initialement) pour les familles XC4000EX, XC4000XL et Virtex
- Implémenté complètement en Java

La possibilité de créer rapidement et dynamiquement des configurations XC4000EX/XL et Virtex a rendu possible l'exploitation des FPGAs Xilinx dans de nouveaux domaines d'utilisation. Les domaines les plus importants sont :

Outils de conception spécifiques : Des outils spécifiques utilisant la reconfiguration dynamique peuvent être construits avec JBits. Ces outils spécifiques pour une application peuvent être assez petits et rapides pour que l'intégration des FPGAs Xilinx soit transparente pour les utilisateurs. En plus, comme JBits est implémenté en Java, le support des outils et bibliothèques Java déjà existants comme par exemple AWT permet un support direct pour la construction de GUI et d'autres tâches de programmation communes.

Outils de conception traditionnels : Comme JBits fournit un modèle et une interface de programmation aux FPGAs Xilinx, JBits peut être utilisé comme une base pour produire des outils de placement-routage traditionnels. Cette possibilité permet aux chercheurs d'effectuer des expériences sur les nouveaux algorithmes en utilisant les dispositifs Xilinx.

Génération des noyaux : Des noyaux pré-placés et pré-routés peuvent être produits avec JBits. Comme ces noyaux sont produits en Java, ils peuvent être paramétrés, embarqués et après utilisés pour produire d'autres noyaux. Et contrairement aux noyaux existants, des noyaux JBits peuvent être paramétrés et initialisés pendant l'exécution. Cela offre beaucoup plus de possibilités que les noyaux statiques, paramétrés pendant la compilation.

Calcul sur support reconfigurable : Les circuits peuvent être produits et modifiés dynamiquement. Cela s'étend de la modification dynamique des paramêtres

et noyaux jusqu'à la conception tout à fait dynamique des circuits. JBits permet un niveau de flexibilité qui n'est pas disponible dans d'autres outils de conception.

Le travail réalisé dans cette thèse est la première utilisation de cet environnement pour réaliser des injections de fautes. Le programme développé à partir de JBits a permis de démontrer la faisabilité des techniques d'injection présentées précédemment, à la fois pour les collages et pour les SEUs.

Résultats : calculs et mesures

Le temps nécessaire pour une expérience d'injection de fautes utilisant la méthodologie proposée peut être décomposé en 3 parties : le temps d'initialisation du dispositif (t_{init}), le temps d'exécution (t_{exec}) et le temps d'injection (ou temps de reconfiguration du FPGA, t_{inject}). Le temps d'initialisation comprend le chargement initial du circuit, c'est-à-dire le chargement du bitstream au début de l'expérience. Le temps d'exécution correspond à l'application de l'horloge au dispositif. Le temps d'injection comprend le temps nécessaire pour l'injection des fautes en reconfigurant (partiellement) le FPGA. La section suivante décrit avec des valeurs numériques ces temps en cas d'injection de fautes de collage et de *SEU*s.

Calculs numériques

Injection de fautes de collage

Considérons un exemple simple : l'injection d'une faute de collage en utilisant un dispositif XCV50 avec une fréquence de configuration et de lecture (*readback*) de 1 MHz. On considère que 10,000 fautes sont injectées ($N_{fautes} = 10000$) et 1000 cycles d'exécution ($N_{cyclesparexprience} = 1000$) sont faits pendant l'expérience d'injection. La reconfiguration et la lecture du dispositif sont faits avec 8 bits en parallèle. Considérons le cas classique d'injection des fautes : une faute est injectée par expérience et une expérience doit être faite sans faute pour servir de référence. Comme cela, le nombre total de cycles peut être calculé selon l'équation suivante :

$N_{cyclestotal} = (N_{fautes} + 1) * N_{cyclesparexprience}$

Le temps d'initialisation (le chargement du bitstream total dans le dispositif) avec ces conditions peut être calculé facilement en sachant qu'un bitstream XCV50 contient 559200 bits :

$t_{init} = \frac{559200}{10^3 * 8} = 69,9ms$

Les temps d'initialisation peuvent être calculés de la même manière pour les différents types de FPGAs.

Pour calculer le temps nécessaire pour injecter (et enlever) une faute, il faut savoir que deux fois la reconfiguration de 8 bits d'un LUT d'un CLB doit être faite pour une faute de collage à 1 ou 0 si la reconfiguration partielle est appliquée. La reconfiguration partielle d'un Virtex est réalisée par des "*frames*" (ou trames), correspondant à la plus petite quantité d'information pouvant être reconfigurée. Une trame donnée contient seulement un bit d'un certain LUT, par exemple le bit d'indice 0 de chaque LUT F de la tranche 0 pour chaque CLB de la colonne considérée. Comme cela, pour reconfigurer (ou lire) 8 bits d'un LUT, 8 trames doivent être reconfigurées (ou lues). Pour injecter puis enlever une faute de collage à 0 ou 1, 8 bits doivent être reconfigurés 2 fois, donc 16 trames doivent être reconfigurées au total.

Une trame d'un dispositif XCV50 contient 384 bits. Comme cela, le temps nécessaire pour injecter une faute de collage peut être calculé :

$t_{injecteruncollage} = \frac{16*364}{10^3*8} = 7,28 * 10^{-1} ms$

Comme 10,000 fautes sont injectées, le temps total d'injection de fautes pour la campagne sera :

$t_{inject} = 10000 * t_{injecteruncollage} = 7280 ms$

Le nombre total des expériences est 10,001 comme décrit ci-dessus, et une expérience contient 1000 cycles. Le nombre total des cycles à exécuter sera donc :

$N_{cycles} = (10000 + 1) * 1000 = 10001000$

Le temps d'exécution peut être calculé facilement (avec une fréquence d'exécution de 1 MHz) :

$t_{exec} = \frac{10001000}{10^3} = 10001 ms$

D'après ces équations, le temps total pour une campagne d'injection de fautes de collage à 1 ou 0 est :

$t_{total} = 69.9 + 7280 + 10001 = 17,350.9 ms$

Similairement aux calculs au-dessus, le temps d'injection peut être calculé pour différentes conditions (par exemple, en faisant varier la fréquence de configuration, le nombre de fautes, etc.).

Injection de *SEU*

Similairement aux fautes de collage, le temps total nécessaire pour injecter un *SEU* peut aussi être évalué. Dans ce cas, l'état de certaines bascules doit être lu et l'état de

certains interrupteurs (déterminant si un *set* ou *reset* est appliqué à la bascule) doit être reconfiguré. Dans le pire cas, fonction du placement des bascules utilisées dans le FPGA, l'état de toutes les bascules et de tous les interrupteurs est concerné. Comme cela, un nombre beaucoup plus important de bits doit être reconfiguré et lu dans le cas des *SEU*s que dans le cas des fautes de collage. Dans le pire cas, toutes les bascules doivent être lues une fois et tous les interrupteurs doivent être reconfigurés deux fois a chaque injection. La lecture d'un type de bascule de toute une colonne nécessite la lecture de deux trames et une commande de lecture doit aussi être envoyée qui est de 28 octets, ce qui donne 992 bits pour un type de bascule d'une colonne. Comme il y a 4 sortes de bascules dans une colonne et 24 colonnes dans un dispositif XCV50, le nombre total des bits à lire pour obtenir les états de toutes les bascules du dispositif sera :

$n_1 = 992 * 4 * 24 = 95232$

La reconfiguration de tous les interrupteurs nécessite la reconfiguration de 11,300 octets. Comme cela doit être fait deux fois, le nombre total de bits devant être reconfigurés quand tous les interrupteurs sont modifiés est :

$n_2 = 11300 * 2 * 8 = 180800$

Considérons le même environnement que dans la section précédente : une fréquence de configuration et de lecture de 1 MHz, un dispositif XCV50 et la configuration ou lecture de 8 bits en parallèle. En premier, on calcule les différents temps en pire cas, c'est-à-dire lorsque toutes les bascules doivent être lues et tous les interrupteurs doivent être reconfigurés. Dans ce cas, le temps nécessaire pour injecter une faute sera :

$t_{injectseu} = \frac{N_{fautes}(n_1+n_2)}{f_{conflecture}N_{bitsparal}}$

En remplacent les paramètres par les valeurs numériques, on peut calculer le temps nécessaire pour injecter un *SEU* en pire cas :

$t_{injectseu} = \frac{(95232+180800)}{10^3*8} = 34,504ms$

Quelques durées de campagnes d'injection sont donnés dans le tableau 1.

Comme indiqué ci-dessus, tous ces calculs sont des calculs pire cas. Il est possible que seulement une partie des interrupteurs/bascules doive être reconfigurée/lue, en fonction du résultat du placement et de l'occupation du FPGA par le circuit. Le pire cas considéré ci-dessus est le cas dans lequel 100% du dispositif est occupé par le circuit ou bien dans lequel les bascules utilisées sont réparties sur l'ensemble du dispositif. De plus, la fréquence de configuration et lecture avait été supposée assez

	10,000 fautes injectées et 1,000 cycles par expérience	**10,000 fautes injectées et 10,000 cycles par expérience**	**1,000,000 fautes injectées et 100,000 cycles par expérience**
Initialisation (ms)	69.9	69.9	69.9
Temps d'éxecution (min)	0.167	1.67	1,666.67
Temps d'injection (min)	5.75	5.75	575.07
Total (min)	5.92	7.42	2,241.74

TAB. 1: Durée d'exemples de campagnes d'injection de SEUs en utilisant la reconfiguration partielle (pire cas)

basse (1 MHz).

Considérons maintenant un meilleur cas : seulement 75% du dispositif est occupé par le circuit, comme cela seulement 75% des bascules et des interrupteurs doivent être lus/reconfigurés, et la fréquence de configuration/lecture est de 10 MHz (les autres paramètres étant inchangés).

Comme cela, le nombre de bits qui doit être lu pour lire les états des bascules sera :

$n_1 = 95232 * 0.75 = 71424$

Similairement, le nombre des bits à reconfigurer pour changer les interrupteurs devient :

$n_2 = 180800 * 0.75 = 135600$

Comme cela, le temps nécessaire pour injecter une faute est réduit à :

$t_{injectseu} = \frac{(71424+135600)}{10^4 * 8} = 2,587ms$

Les durées de campagnes correspondant sont résumées dans le tableau 2.

	10,000 fautes injectées et 1,000 cycles par expérience	**10,000 fautes injectées et 10,000 cycles par expérience**	**1,000,000 fautes injectées et 100,000 cycles par expérience**
Initialisation (ms)	69.9	69.9	69.9
Temps d'éxecution (min)	0.167	1.67	1,666.67
Temps d'injection (min)	0.43	0.43	43.12
Total (min)	0.60	2.10	1,709.79

TAB. 2: Durée d'exemples de campagnes d'injection de SEUs en utilisant la reconfiguration partielle (cas plus favorable)

En comparant les tableaux 1 et 2 on peut voir qu'un gain d'un ordre de grandeur sur la durée d'une campagne peut être atteint assez facilement quand les conditions ne sont pas celles du pire cas (sans supposer le meilleur cas).

Une étude de sensibilité des différents paramètres a été réalisée et a permis de mettre en évidence en particulier l'influence du nombre de bits à reconfigurer sur l'efficacité de l'approche proposée. Ceci conduit à conclure à la mauvaise adéquation de l'architecture Virtex pour ce type d'application.

Des comparaisons avec les techniques d'injection préalablement connues (à base de simulation ou d'émulation) montrent que l'approche proposée est nettement plus efficace que la simulation, et peut dans divers cas être compétitive, en terme de durée de campagne, avec une émulation classique. Elle a en plus divers avantages, et permet en particulier de réduire le temps de préparation de la campagne. L'efficacité, par rapport à une émulation classique, dépend cependant fortement du nombre de fautes à injecter par rapport au nombre de cycles de chacune des expériences d'injection.

Mesures expérimentales

Des mesures expérimentales ont été réalisés sur la carte XSV50 [142] qui a été utilisée pour démontrer la faisabilité de l'approche. Cette carte de développement (et l'interface XHWIF nécessaire pour JBits) ne sont pas conçues pour la réalisation de reconfigurations à haute fréquence ; le chargement ou la lecture du FPGA est fait par le port paralléle qui fonctionne à une vitesse assez basse (2Mbits/sec) et la fréquence de configuration/lecture de la carte á l'aide des interfaces JBits et XHWIF était seulement de 3.9 kHz. Avec ces conditions expérimentales, il n'est pas possible d'obtenir des résultats tels que décrits dans les sections précédentes. Pour atteindre de tels résultats, un environnement spécifique pour l'injection de fautes devrait être conçu, afin de pouvoir réellement exploiter les possibilités de l'approche proposée.

Des mesures expérimentales concernant l'injection des *SEU*s sont résumées dans le tableau 3. Si on les compare avec les résultats des tableaux 2 et 3, on peut voir que les valeurs mesurées sont beaucoup plus élevées que les valeurs calculées dans les sections précédentes à cause des limitations en vitesse de la carte.

Initialisation (ms)	**Injection de faute (ms)**	**Exécution (ms)**
19,678	3,566	55,961
19,688	3,575	55,873
19,718	3,498	55,912

TAB. 3: Mesures expérimentales d'injection de *SEU* sur la carte XSV50, pour trois séries d'expériences

Conclusion et perspectives

Le sujet de la thèse est le développement d'une nouvelle méthodologie d'injection de fautes, en utilisant le prototypage matériel et la reconfiguration partielle. La faisabilité de l'approche a été démontrée pour deux sortes de fautes : les fautes de collage et les *SEU*s. L'application a été réalisée en utilisant un FPGA Virtex de Xilinx, et l'interface JBits qui rend possible la reconfiguration partielle du dispositif.

Les calculs montrent que le temps d'injection d'une faute de collage est de l'ordre de la milli-seconde, et que le temps d'injection d'un *SEU* est de l'ordre de quelques millisecondes avec l'environnement utilisé comme démonstrateur.

Une campagne d'injection de *SEU*s avec la méthodologie proposée ne serait pas forcément plus rapide qu'une campagne en appliquant une méthodologie traditionnelle (prototypage matériel classique). Le temps nécessaire dépend de plusieurs facteurs, qui seraient à optimiser pour ce type d'application et qui ont été identifiés lors de l'étude. Un environnement adapté doit assurer un accés à haute vitesse au FPGA pour pouvoir profiter des caractéristiques technologiques (recofiguration/lecture rapide) du dispositif. Le placement du circuit dans le FPGA doit être optimisé pour réduire au minimum le nombre de trames à reconfigurer. Potentiellement, d'autres architectures de FPGA (par exemple, AT6K, AT40K [11] [14]) mieux adaptées que l'architecture Virtex, pourraient aussi être considérées.

La réalisation d'un tel environnement optimisé pour les injections de fautes est une perspective de poursuite du travail.

Magyar nyelvű összefoglaló

Bevezetés

A dolgozatban bemutatott kutatómunka tárgya a futási-idejű (és parciális) újrakonfigurálásnak a hibainjektálás területén lehetséges alkalmazási lehetőségeinek vizsgálata.

A hibainjektálási technikák alkalmazásának célja egy rendszer biztonságosságának validálása abban az esetben, ha hiba lép fel a rendszerben. Ennek módja, hogy szándékosan hibá(ka)t helyezünk el a rendszerben, majd analizáljuk a rendszer (és alkotóelemei) működését. Az újrakonfigurálható hardver eszközök (pl. FPGA-k) alkalmasak rendszer-prototípusok implementálására és tesztelésére; a rendszer magasszintű leírása szintézis és huzalozás-lista generáló algoritmusok futtatása után letölthető az FPGA eszközbe. A prototípus-futtatás tehát *in-system* hardver-emulációt tesz lehetővé az áramkör gyártása előtt. A hibainjektálásnak hardver prototípusban való alkalmazása megfelelő módszer lehet a biztonságosságnak a tervezési folyamat elején való vizsgálatára.

A futási-idejű újrakonfigurálás (*Run-Time Reconfiguration* - RTR) egy olyan módszer, amely során a hardvert az alkalmazás futtatása közben konfiguráljuk újra. Az RTR technikák különbözőek lehetnek annak függvényében, hogy egy újrakonfigurálási lépésben az egész eszközt újrakonfiguráljuk vagy csak egy részét. A dolgozatban a parciális újrakonfigurálást javasolom a hibainjektálási eljárások felgyorsítására.

Újrakonfigurálás és fajtái

A *Reconfigurable computing* kifejezés azt jelenti, hogy lehetséges egy rendszer architektúrájának valósidejű megváltoztatása. Ezek a rendszerek leginkább ebben a tulajdonságukban különböznek a fix hardvertől (pl. mikroprocesszorok).

Az újrakonfigurálható hardver eszközökben 3 fő típust különböztethetünk meg:

- logika (FPGA-k)

- beágyazott rendszerek (pl. újrakonfigurálható koprocesszorok)
- számítógépek (teljesen újrakonfigurálható platformok amelyek FPGA eszközöket (is) tartalmaznak

Az újrakonfigurálás technikájának a (mi szempontunkból) legfontosabb fajtái a fordítási-idejű újrakonfigurálás (*Compile-Time Reconfiguration* - CTR) és futási-idejű újrakonfigurálás (*Run-Time Reconfiguration* - RTR) [64]. CTR-nél minden applikáció egy konfigurációból áll, amely maximális mérete az adott újrakonfigurálható áramkör méretével megegyezik. RTR technika esetén egy-egy alkalmazás több konfigurációból áll, amelyek különböző időpontokban vannak letöltve az eszközbe, és egymással adatcserét, kommunikációt folytatnak. Ezen két technikát mutatja be a következő két fejezet.

Fordítási-idejű újrakonfigurálás (CTR)

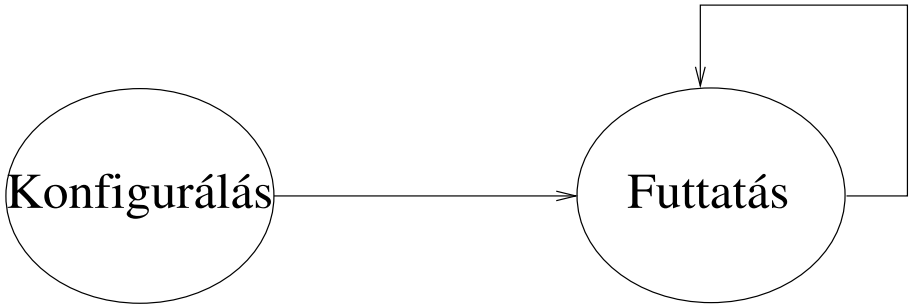
Alkalmazások újrakonfigurálható áramkörökben való implementálásánál a legegyszerűbb és legelterjedtebb technika a CTR alkalmazása [64]. A CTR alkalmazások legfontosabb jellemzője hogy egyetlen konfigurációból állnak. Ezt az egyetlen konfigurációt töltik le az FPGA eszközbe az alkalmazás futtatása előtt, és ezután a futtatás végéig ez a konfiguráció marad az áramkörben (8. ábra).

Ez a technika igen hasonló az ASIC eszközök alkalmazásához: a futtatandó alkalmazás szempontjából nincs jelentősége, hogy FPGA vagy ASIC eszközt használunk. Az FPGA-k előnye az árban rejlik: ugyanazt a hardvert több alkalmazás futtatására is fel lehet használni. Ezzel szemben az FPGA eszközök lassabbak, mint az ASIC áramkörök.

A legfontosabb hasonlóság tehát a hagyományos hardver-tervezés és a CTR-tervezés között a hardver-allokáció statikus volta: a hardver statikus az egész applikáció futtatása alatt. Az FPGA technológia lehetővé tenne dinamikus megközelítést, a CTR alkalmazások azonban statikus módszereken alapszanak, miután a tervezési folyamat így a legegyszerűbb.

Futási-idejű újrakonfigurálás (RTR)

A CTR alkalmazásokkal ellentétben az RTR alkalmazások dinamikusan allokálják a logikai erőforrásokat a futtatás egész ideje alatt. Ennek megfelelően minden alkalmazás több konfigurációból áll, ezek a konfigurációk csak az applikáció egy-egy részét



8. ábra. Fordítási-idejű újrakonfigurálás (CTR)

implementálják. Az RTR applikációk tehát többször konfigurálják újra az FPGA eszközt az applikáció futtatása alatt [64].

A hardver dinamikus tulajdonsága két új problémát vet fel. Az első az applikáció, algoritmus partícionálása olyan szegmensekre amelyeknek nem kell egy időben futniuk vagy nem is futhatnak egy időben (időbeli partícionálás). Ezen szegmensek részeit konkurens módon futtatjuk, és a szegmenseket egymás után, szekvenciálisan töltjük le az FPGA-ba. Ezt az időbeli partícionálást a legegyszerűbben a működés különböző fázisai alapján lehet a legegyszerűbben elvégezni. Minden fázist hozzárendelhetünk egy szegmenshez, vagyis egy FPGA-konfigurációhoz.

Az RTR ötlete az erőforrás-igények miatt vetődött fel. Nagy méretű alkalmazásokat nem feltétlenül lehetséges statikus módon implementálni újrakonfigurálható hardverben mivel az eszköz mérete nem teszi lehetővé (amikor az RTR ötlete megszületett, az FPGA eszközök mérete jóval kisebb volt mint napjaink eszközei, lásd Virtex-II vagy Virtex-E család). Miután az RTR-nél az alkalmazásnak mindig csak egy része van az áramkörben, az erőforrás-igény alacsonyabb. Ez azonban egy tervezési problémát vet fel, miután a jelenlegi tervezőrendszerek nagyrészt statikus tervezési modellre készültek. A másik probléma RTR alkalmazások esetén a konfigurációk közötti kommunikáció és adatcsere vezérlése: egyes konfigurációk által kiszámolt részeredményekre más konfigurációknak szükségük van. Az egyes konfigurációkat ennek megfelelően nagyon körültekintően kell megtervezni, hogy az adatcsere és a kommunikáció ne sérüljön közöttük.

Két alapvető RTR technikát különböztethetünk meg: Globális és Lokális (másnéven parciális). A két módszer között a különbség az erőforrások allokálásának módjában van. A következő két alfejezet a Globális illetve a parciális RTR-t tekinti át röviden.

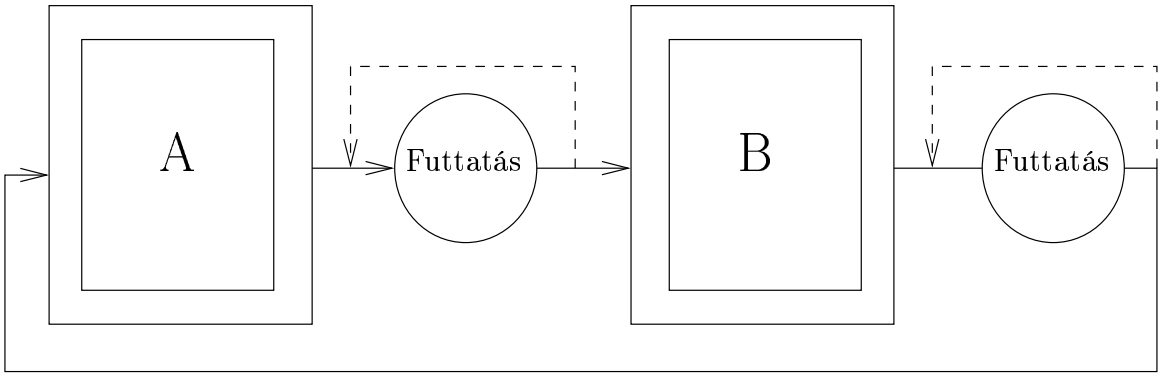
Globális RTR

Globális RTR-nél minden egyes újrakonfigurálási lépésben a teljes eszközt újrakonfiguráljuk. Ennek megfelelően az applikációt időbeli fázisokra lehet partícionálni, és minden egyes fázis megfeleltethető egy-egy konfigurációnak, amely a teljes eszközt lefedi (9 ábra). Globális RTR-t alkalmazó applikációknál lehetőleg azonos méretű partíciókat kell létrehozni az erőforrások minél jobb kihasználtsága miatt [64]. Egyértelmű, hogy ez a lépés iteratív, miután egy partíció méretét nehéz megbecsülni a szintézis és a huzalozási algoritmusok futtatása előtt. Szükség van egy kiindulási partícióra, majd iterációt kell végezni mind a terven, mind a partíción, egészen addig amíg többé-kevésbé azonos méretű partíciókat kapunk. Itt problémát jelent az, hogy nehéz megbecsülni, szimulálni hogy egyik partícióból milyen másik partíció lesz bizonyos tervbeli változtatások után. A legfőbb hátránya a Globális RTR-nek viszont az, hogy a nagyjából azonos méretű partíciók létrehozása nagyon komplex feladat. Amennyiben nem tudjuk ezt a feladatot megoldani, az FPGA eszköz erőforrásainak kihasználása nem lesz megfelelő.

A partícionálás után a következő feladat a konfigurációk közötti kommunikáció megvalósítása. Miután minden fázis az összes erőforrást lefoglalja, a különböző konfigurációk interfészei lehetnek fixek, és minden modult ugyanabban a kontextusban lehet megtervezni. A legegyszerűbb módja a kommunikációnak, ha az egyes konfigurációk az eredményeiket, kimenő adataikat működésük befejezése után egy meghatározott területre mentik el (pl. memóriaelem, regiszter stb.). A következő konfiguráció erről a helyről olvassa majd be ezeket az adatokat, majd ezek a lépések ismétlődnek az alkalmazás futásának végéig.

A Globális RTR legfontosabb előnye, hogy viszonylag egyszerű. Miután az alkalmazásukat sikerült megfelelően partícionálni, már mindegyik partíció kezelhető úgy mint egy független modul, és bármilyen fejlesztőrendszerrel végezhetünk optimalizálást az egyes konfigurációkon.

Több olyan applikációt is kifejlesztettek már ami Globális RTR-t használ. Egyike az elsőknek az RRANN [51] [52] [53], amelyik a neurális hálózatok területén alkalmazza az RTR-t. Ezen kívül sok más applikáció létezik, amely fejlesztésénél RTR-t alkalmaztak.



9. ábra. Globális RTR

Lokális (parciális) RTR

A Lokális (parciális) RTR egy a Globális RTR-nél is flexibilisebb módszer. Amint a neve is mondja, a parciális RTR applikációknál szelektív módon az erőforrások egy részhalmazát konfiguráljuk újra minden egyes konfigurációs lépésben (10 ábra). Néhány FPGA-nak szinte bármelyik részét lehetséges parciálisan újrakonfigurálni, ennek megfelelően finomabb erőforrás-allokáció valósítható meg, mint Globális RTR-rel. Csak az eszköz azon részeit konfiguráljuk újra, amelyekben változtatásokat akarunk végrehajtani, ennek megfelelően a konfigurálásra fordított idő csökken, valamint hatékonyabb erőforrás-kihasználás valósítható meg. A Lokális RTR egy másik előnye, hogy lehetséges csak az alkalmazás dinamikus részeinek az újrakonfigurálása, míg a statikus részek változatlanok maradnak.

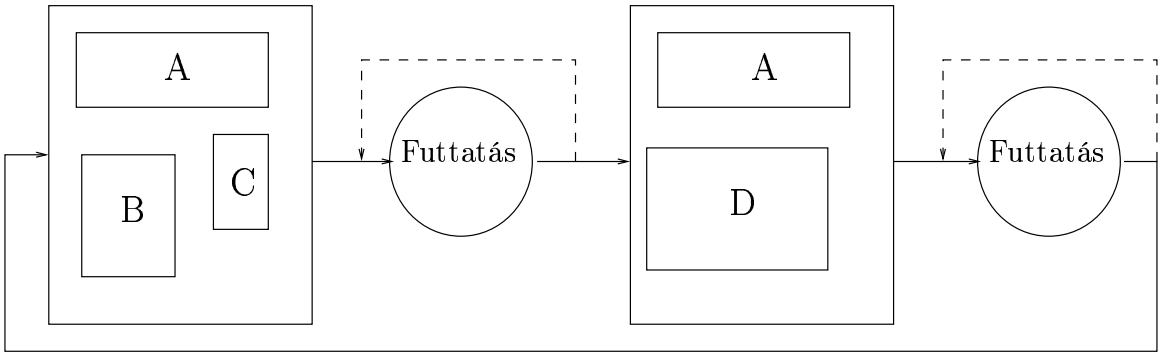
A parciális RTR-rel készített applikációk inkább funkcionális partícionáláson alapszanak, ellentétben a Globális RTR időbeli partícionálásával [64]. A parciális RTR partíciói közül egyszerre több is lehet aktív (betöltve az FPGA-ba), ellentétben a Globális RTR partícióival amelyek közül egyszerre mindig csak egy aktív. A Lokális RTR partíciói mind független modulokként kezelhetőek, amelyeket igény szerint töltünk le az FPGA eszközbe. Ezen partíciók közül egyszerre többet is letölthetünk az FPGA eszközbe, és egy-egy partíció bármennyi erőforrást lefoglalhat az eszközből.

A parciális RTR legfőbb előnye a Globális RTR-rel szemben az erőforrások jobb kihasználása, főleg olyan applikációk esetén amelyeket nehéz időben egymást követő és nagyjából azonos méretű partíciókra osztani. Sajnos ez az előny azonban egy nehézséget is magában hordoz: a komplexitás növekedését. Lokális RTR-nél az egyes partíciók közötti interfészek nem fixek, sőt minden egyes konfiguráció esetén különbözhetnek. Ennek megfelelően az interfészek vezérlésére nagy gondot kell fordítani

(főleg, amikor egy időben több konfigurációt töltünk az eszközbe) hogy a konfigurációk közötti kommunikáció ne sérüljön. Emellett a huzalozási algoritmusokra is ügyelni kell a konfigurációk közötti helyes kommunikáció érdekében. A processzek helyes időzítése miatt a hardver/szoftver szétválasztást is figyelmesen kell végezni a tervezési folyamat elején [72].

A jelenlegi fejlesztőrendszerek sajnos nagyrészt nem támogatják a parciális RTR-t: sem a konfigurációk közötti adatáramlás, kommunikáció szimulálását, sem a huzalozási algoritmusok vezérlését. Egyike a kevés fejlesztőrendszereknek, amely támogatja a parciális RTR-t a Xilinx által kifejlesztett JBits, amelynek segítségével a dolgozat kísérletei készültek.

Egyike az első fejlesztéseknek, amely parciálsi RTR-t alkalmaz, az RRANN-2 rendszer, amely neurális hálózatok területén alkalmazza ezt a technikát, valamint a DISC (Dynamic Instruction Set Computer) [140].



10. ábra. Lokális (parciális) RTR

Hibainjekálás

Ma már nem csak missziókritikus rendszerekben, hanem egyéb applikációkban is fontos, hogy hiba esetén tovább működjenek. Ezért ezekbe különböző hibatűrési mechanizmusokat terveznek. Ahhoz azonban, hogy a hibatűrési mechanizmusok működőképességét ellenőrizzük természetesen hibás futásra van szükség, ezeket azonban mesterségesen kell előidézni, hiszen egyrészt az adott hibák várhatóan ritkán fordulnak elő, másrészt nem minden hibaok lép fel természetes módon.

A hibainjektálás egy olyan technika, amely számítógépes rendszerekben hardverilletve szoftver tesztelésére egyaránt használható. A technika lényege, hogy különböző

módon szándékosan hibákat helyezünk el a rendszerben, majd ezt követően futás közben figyeljük a rendszer működését, elsősorban hibák (fault), hibás rendszerállapot (error) vagy hibajelenség (failure) keletkezése szempontjából. A hibainjektálás célja mind hardver, mind szoftver rendszerekben a rendszer hibatűrésének a kísérleti ellenőrzése.

Alkalmazási körét tekintve a hibainjektálás sokféle rendszerben és sokféle módon alkalmazható:

- modellen szimuláció segítségével
- működő prototípuson
- használatban lévő rendszeren.

Ilyen módon a különböző kölcsönhatások okozta gyengeségek ismerhetőek meg. Általában a hibainjektálást inkább azon célból alkalmazzák, hogy megvizsgálják egy hibatűrő / beágyazott rendszer ismert hibákkal szembeni védettségét, és ilyen módon adatokat szerezzenek a hibatűrő rendszerbe épített mechanizmusok védelmi hatékonyságáról.

A hibainjektálás, vagyis mesterséges hibák elhelyezése egy szimulált vagy valós, tesztelés alatt álló számítógépes rendszerben lényegében három főbb előnnyel szolgál:

- megismerhetjük, megérthetjük a valós hibák hatásait,
- visszajelzést kapunk a rendszerbe épített hibatűrő mechanizmusok hibafedéséről,
- előrejelzést, becslést adhatunk a rendszer várható viselkedését illetően.

Ennek megfelelően a hibainjektálási technikák segítséget nyújtanak a számítógépes rendszerek megbízhatóságának feltérképezésében.

Hibainjektálás célja, áttekintés

A missziókritikus számítógép-vezérelt rendszerek meghibásodása beláthatatlan következményekkel járhat. Az ilyen meghibásodások emberéleteket követelhetnek több területen is: légi közlekedésben, vasúti irányításban, egészségügyi ellátásban, ipari intézmények vezérlésnél, nukleáris objektumok irányításánál vagy katonai létesítményeknél, hogy csak pár példát említsünk. Pár perces kiesés a működésben óriási

méretű gazdasági veszteséget okozhat az üzleti élet minden szintjén, de elsősorban a távközlésben, a bankvilágban, biztosító-társaságok esetében, és minden olyan ipari létesítménynél, amelyik napi 24 órában, évi 365 napon át működik.

Már hosszú idő óta nyilvánvaló, hogy az ilyen rendszerek szolgáltatásbiztonsága nem garantálható önmagában pusztán elővigyázatos tervezéssel és minőségbiztosítással, hanem védelmi mechanizmusokat és más hibaelhárítási módszereket kell beépíteni. A hibatűrés lényege, hogy a számítógépes rendszernek akkor is az előírt módon kell működnie, amennyiben hiba lép fel a rendszerben. Ez a működés jelentheti a szolgáltatás hibátlan folytatását, vagy pedig biztonságos állapotba lépést.

Egy hibatűrő rendszert viszont, mielőtt üzembe helyeznék, tesztelés alá kell vetni, valamint működését verifikálni és validálni kell. Ennek segítségével megállapíthatóak az alapvető kritériumoknak való megfelelés, valamint megválaszolható:

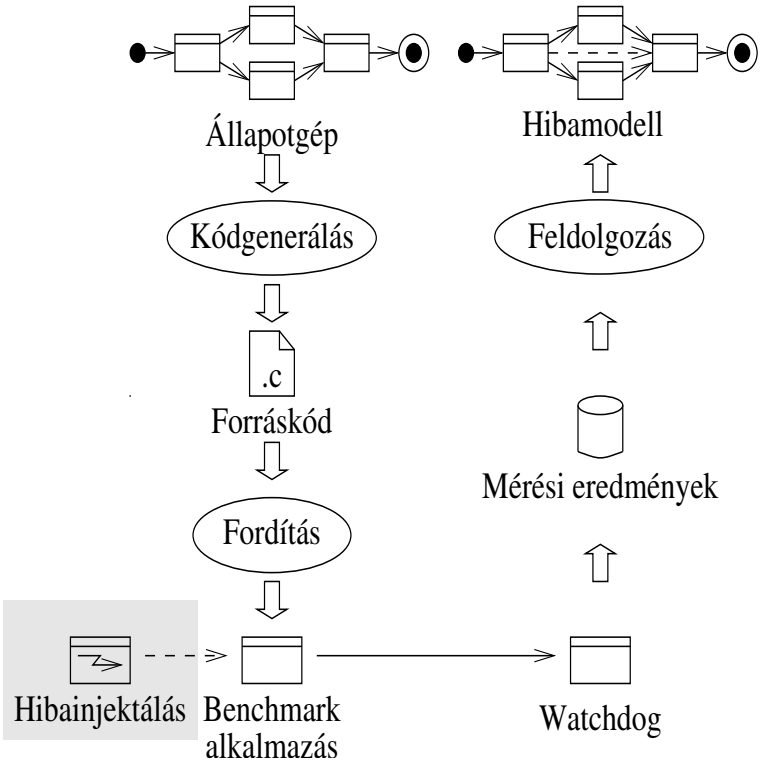
- a rendszer kilábal-e a hibás állapotából,
- minden hibafajtával szemben hibatűrő vagy csak bizonyos hibákkal szemben,
- minden esetben vagy csak bizonyos feltételek mellett hibatűrő,
- vannak-e a rendszernek kritikus részei, amelynek meghibásodásai minden esetben az egész rendszer leállását, hibás működését vonják maguk után

és még sok hasonló kérdés.

A válaszok e kérdésekre rendkívül fontosak mind a hibatűrő rendszerek tervezői, mind használói számára. A rendszerek gyártói adott költségráfordítással szeretnének jó szolgáltatást elérni hogy ez piacon minöségi előnyt jelenthessen, a felhasználók pedig a számára nyújtott szolgáltatás minőségét akarja garantálva látni.

A hibainjektálás folyamatát nagy vonalakban az 11 ábra mutatja [112].

A hibainjektálással kapcsolatban kiemelten fontos az alkalmazási körének körülhatárolása. Eleinte úgy gondolhatnánk, a hibainjektálás alkalmas lehet rendszerek megjavítására, különböző hiányosságainak feltérképezésére. A hibainjektálás annyiban képes a valóságot tükrözni, amennyiben az alapul vett hibamodell megfelel a valóságos hibáknak. Jellegzetesen a hibainjektorok a technológiai tapasztalatokból vonnak ki egy általánosított hibamodellt, de új technológiák esetén gyakran fellépnek olyan újszerű hibaforrások, amelyeket a korábbi hibainjektorok nem tudnak. Mindebből az következik, hogy a hibainjektálás nem kifejezetten alkalmas a rendszernek



11. ábra. A hibainjektálás folyamata

önmagának fejlesztésére, javítására, ellenben annál inkább alkalmas a rendszer hibatűrő tulajdonságainak tesztelésére [137]. Ennek megfelelően egy ismert hibát injektálhatunk a rendszerbe, majd figyelhetjük, hogy a rendszer kimenetei a várt értékeket veszik-e fel vagy sem.

A fentieknek megfelelően a hibainjektálásnak két ajánlott felhasználási területét állapíthatjuk meg. Az első terület a rendszer validációja. Amennyiben egy rendszert olyan módon terveztek, hogy bizonyos hibaosztályokat tűrjön, vagy pedig egyes rendszerállapotok legyenek kizárva bizonyos hibák esetén, abban az esetben ezen hibák közvetlenül injektálhatóak a rendszerbe hatásaik vizsgálata céljából. A rendszer vagy helyesen fog működni vagy nem, hibatűrése pedig ennek megfelelően kerül mérésre [8]. Némely, kifejezetten nehezen tesztelhető rendszer esetén, amelyeknél a hibák olyan ritkán jelentkeznek, hogy nem lehetséges a rendszert működési környezetében tesztelni, a hibainjektálás hatékony eszköz lehet ezen hibák előfordulásának elősegítésére és annak ellenőrzésére, hogy a rendszer helyesen működik-e.

Ezen kívűl alkalmazzák a hibatűrést annak a vizsgálatára is, hogy egy nem hibatűrőre tervezett rendszerben mi történik. A hibainjektálásnak ezen második területen történő alkalmazásához két komolyabb nehézséget kell kiküszöbölnünk. Az első a

rendszerek sokrétű működésében és lehetséges meghibásodási módjaikban rejlik. Két rendszer relatív robusztusságának meghatározása nagyon nehéz, hacsak nem ugyanazon feladat végrehajtása közben hasonlítjuk össze működésüket. Ennek megoldását jelentené egy jó metrika létrehozása. A másik fontos kérdés, hogy a metrika mire irányuljon. Az általánosan elfogadott gyakorlat az, hogy a tesztek-hibainjektálások eloszlása megegyezik a hibák előfordulásának eloszlásával a valós rendszerekben. Ellenben ha a rendszer működését valóban szokatlan szituációkban szeretnénk vizsgálni, a teszteket inkább a kevésbe gyakran előforduló körülmények mellett kellene végezni [136]. A szakemberek abban egyetértenek, hogy a hibainjektálás alkalmas lehet a robusztusság metrikájának létrehozására. A metrika megvalósításának konkrét mechanizmusai azonban még nincsenek teljesen kidolgozva.

VHDL modellek és hardver prototípusok

Az eddig alkalmazott hibainjektálási technikák többsége esetén a hibák injektálása a már kész rendszerbe vagy áramkörbe történik. Ilyen technika pl. a lábszintű hibainjektálás, memória-tartalom megváltoztatás [102], nehézion-sugárzás, tápfeszültségmegváltoztatás [82], lézeres hibainjektálás [120] vagy szoftverbe történő hibainjektálás [76].

Ezen technikák kifejlesztésénél később többen is végeztek kutatást a hibainjektálásnak a tervezési folyamat korai fázisaiban történő alkalmazásával kapcsolatban. A legfontosabb megközelítés azon alapszik, hogy az áramkör vagy rendszer magasszintű modellébe (általában VHDL leírásokba) injektáljuk a hibákat. Ilyen módszert alkalmaz pl. [47] ahol mikroprocesszoros rendszerek viselkedési VHDL-leírásaiban végeznek hibainjektálást. [71] majd később [131] vagy [23] olyan technikát mutat be, ahol az áramkör több absztrakciós szintű VHDL-leírásába injektálják a különböző típusú hibákat. Gyakori technika a szimuláció alkalmazása a hibáknak az áramkör viselkedésére gyakorolt hatásának vizsgálatára (pl. [47]). Mint arra [88] is rámutat, a szimuláción-alapuló technikák legnagyobb hátránya a kísérletek elvégzéséhez szükséges nagy idő, főleg több hiba injektálásának esetén.

A szimulációk hosszú futási idejének csökkentése érdekében javasolták hardverprototípusok alkalmazását hardver-emulátor vagy FPGA-alapú újrakonfigurálható rendszer formájában [87]. Az emuláció egy másik előnye, hogy a tervező az áramkör működését, a valósidejű interakciókat egy a végső alkalmazáséhoz közeli környezetben tudja vizsgálni [22]. Hardver emulátor alkalmazása esetén a kezdeti VHDL leírásnak

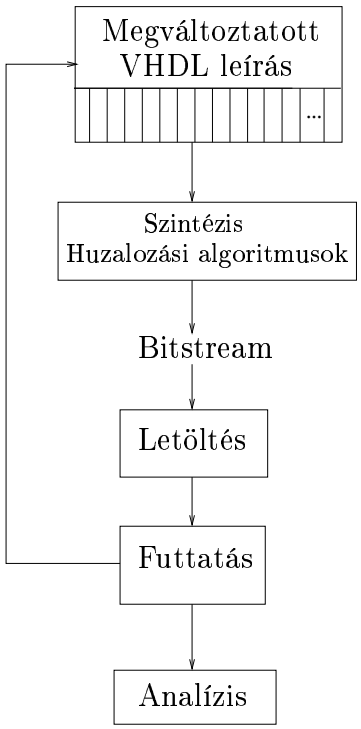
természetesen szintetizálhatónak kell lennie. Néhány, korlátozott esetben a hibaemulálásnál (*fault grading*) alkalmazott módszert használni lehet hardver-emulátoros hibainjektálás esetén (pl. [38], [139]). Ezek a technikák általában csak leragadási hibák (*stuck-at fault*) injektálására korlátozódnak . A SimExpress emulátor speciális esetében a hibákat lehetséges közvetlenül az áramkörbe injektálni, kihasználva az emulátor némely beágyazott funkcióját [1]. Ezek a funkciók azonban a kereskedelemben is rendelkezésre álló hardver emulátorokban nincsenek meg, ráadásul ezek a technikák is a leragadási hibákra korlátozódnak. Az esetek többségében a hibainjektáláshoz szükséges változtatásokat a kezdeti (legmagasabb szintű) leírásban kell elvégezni, figyelve arra hogy a leírás a változtatások után is szintetizálható maradjon, valamint több, a hardver emulátor által támasztott megszorításnak is meg kell felelnie [88]. Ennek megfelelően a szükséges változtatások eszközölése nem egyszerű, ráadásul gyakran több megváltoztatott leírást is generálni kell, amelyek közül mindegyik egy-egy hibahalmaz injektálását végzi el, teszi lehetővé. Ebben az esetben a hardver emulátort többször is újra kell konfigurálni ami egy igen időigényes folyamat, és ennek megfelelően csökkenti a szimulációhoz viszonyított időbeli nyereséget.

A doktori kutatás célja egy új hibainjektálási módszer létrehozása amely rendelkezik az emuláció bizonyos előnyeivel, a hátrányai közül pedig lehetőleg minél kevesebbel. A dolgozatban bemutatásra kerülő technikák egyik előnye, hogy a kezdeti leírást nem kell megváltoztatnunk ahhoz a hibainjektáláshoz. Egy másik (lehetséges) előny a konfigurálási idő csökkentése egy hibainjektálási kísérletsorozat esetén (miután nincsenek különböző módosított VHDL-leírások amelyeket egyesével kellene letölteni az FPGA eszközbe). Emellett a konfigurálási id[a parciális újrakonfigurálás alkalmazásával is csökkenthető.

Egy új hardver emulátoros hibainjektálási technika

A "klasszikus" hardver-emulátort alkalmazó hibainjektálási eljárások során a hibákat a VHDL kódban történő módosításokkal injektálják. Ezek után szintetizálják a kódot, majd egy kapuszintű leírást (*netlist*) generálnak. A huzalozási algoritmusok futtatása után a bináris formájú konfiguráció (*bitstream*) letölthető az FPGA eszközbe. Végül az applikációt futtatják az eszközben és analizálják az injektált hibák esetén történő futtatás eredményeit.

Ennek a hibainjektálási módszernek a folyamatábráját mutatja a 12 ábra.

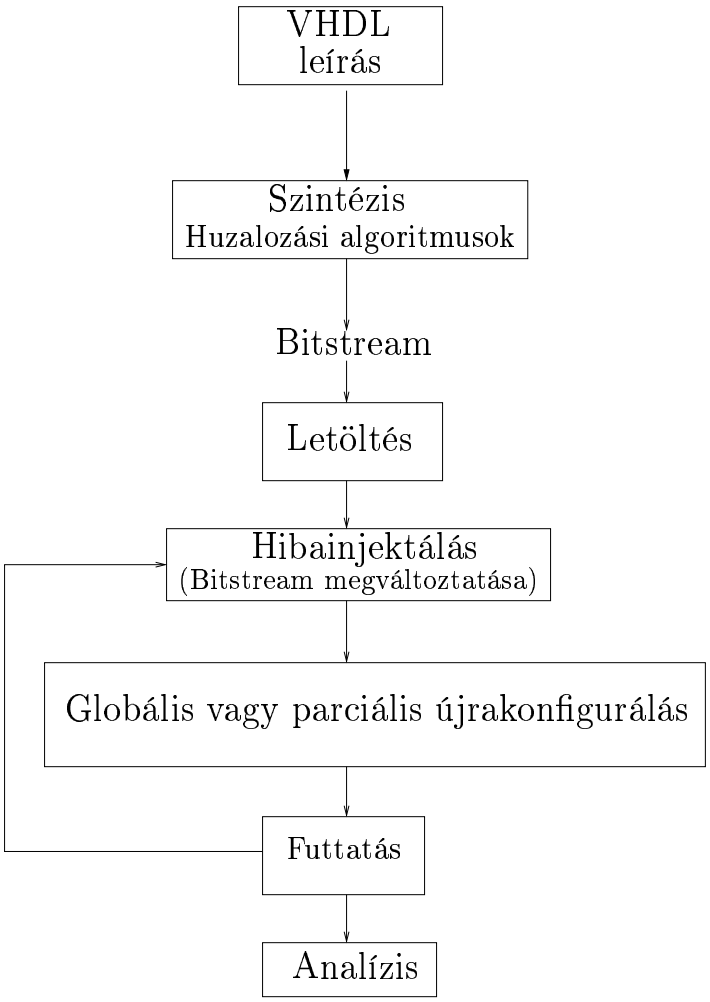


12. ábra.
Hibainjektálás a kezdeti VHDL leírás megváltoztatásával és hardver-prototípus alkalmazásával

Ebben a technikában gyakran szükség van több leírás generálására (pl. az emulátor korlátai mint kapuszám vagy lábszám miatt), amelyek különböző hibainjektorokat tartalmaznak, strukturálisan azonban nagyon hasonlóak. Ezeket a leírásokat különkülön szintetizálják, majd töltik le és futtatják az FPGA eszközben, legvégül pedig analizálják a futtatás eredményeit különböző hibák injektálása esetén. (5 ábra).

A doktori kutatás folyamán kifejlesztett technika a hibák alacsony szinten történő injektálásán alapszik: a hibákat nem tartalmazó (kezdeti) leírást szintetizáljuk, *bitstream*et generálunk amit letöltünk az eszközbe, és a hibákat közvetlenül az eszközbe injektáljuk (parciális) RTR alkalmazásával, az applikáció futtatása közben. A hibákat magába a *bitfile*ba is injektálhatjuk. Ennek megfelelően a kezdeti leírást nem szükséges újraszintetizálni és a huzalozási algoritmusokat újrafuttatni minden egyes hibainjektálási kísérlethez, csak újra l ekell tölteni az FPGA eszközbe. Ennek a módszernek a folyamatábráját mutatja a 13 ábra.

A következő fejezetekben a kifejlesztett módszer implementációjának valamint az implementációhoz használt JBits interfész leírása található.



13. ábra. A kifejlesztett hibainjektálási technika

A módszer implementálása

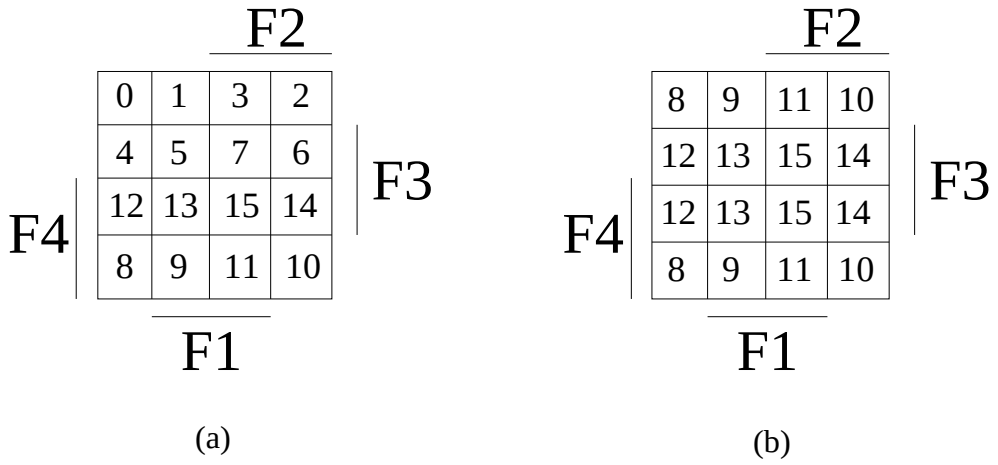
Amint azt fentebb már említettük, a kifejlesztett módszer segítségével kétfajta hiba injektálása lehetséges: leragadási és *SEU*. A következő két fejezet ezen két hibafajta injektálására Xilinx Virtex FPGA eszköz alkalmazásával kidolgozott módszereket mutatja be röviden.

Leragadási hiba injektálása

Egy VHDL leírás szintézise és FPGA-ba történő huzalozása során különböző logikai függvényeket töltünk le az eszközbe. Egy logikai függvényt az FPGA-ban egy *Look-Up Table*-be (LUT) tudunk letölteni, amelyek a Konfigurációs Logikai Blokkokban (*Configurable Logic Block*, CLB) találhatóak. A Xilinx Virtex családban egy LUT-nak 4 bemenete van (a VHDL leírás vezetékei ezekre a bemenetekre képeződnek le),

ennek megfelelően a LUT-ba leképzett logikai függvényt 16 memóriabit határozza meg.

A módszer lényege, hogy a LUT-ban tárolt bináris értékeket megváltoztatjuk, így imitálva a LUT egyik bemenet'enek leragadását. A 14 ábra egy példát mutat egy leragadási hiba injektálására. A LUT Karnaugh-táblaként van ábrázolva.



14. ábra.
Egy 1-be ragadási hiba injektálása az F4 bemenetre: (a) eredeti LUT konfiguráció (b) módosított LUT

Ez a példa az F4 bemenet 1-be ragadását mutatja. A 0-ba ragadási vagy invertálási hibák hasonló módon valósíthatóak meg.

SEU injektálása

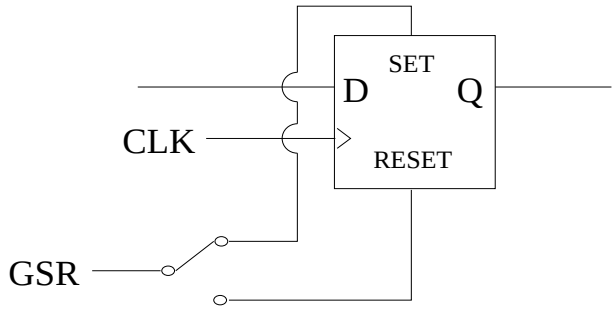
Egy áramköri leírásnak Virtex FPGA-ba történő implementálása esetén az áramkör kombinációs részeit a LUT-okban kerülnek implementálásra. A CLB-k másik fontos alkotóelemei a flip-flopok amelyekbe a különböző regiszterek leképzése történhet. Egy *Single Event Upset* (*SEU*) hiba egy ilyen memóriaelem értékének aszinkron módon történő megváltozását jelenti, 0-ból 1-be vagy 1-ből 0-ba.

A kifejlesztett módszerben a *SEU*-k injektálása a flip-flopok állapotának ellenkezőjére változtatásával történik. A jelenleg rendelkezésre álló környezetben ez csak egyetlen módon valósítható meg: egy *set* vagy *reset* impulzus adásával a flip-flop aszinkron vezérlő bemenetére, a pillanatnyi állapotától függően.

A kísérletekben használt Virtex FPGA családnál az eszköznek egy GSR bemenete van (*Global Set/Reset*) ami be van kötve az FPGA minden lényeges alkotóelemének bemenetére, így a flip-flopok bemenetére is. A set/reset bemenet kétfajta polaritású

lehet: *set* vagy *reset*.

Ezt a legegyszerűbben egy kapcsoló segítségével ábrázolhatjuk: a kapcsoló a flip-flop *set* és *reset* bemenetei között kapcsolja a GSR bemenő vonalat. A 15 ábrán ez van ábrázolva.



15. ábra. Egy Virtex FPGA CLB-jében lévő flip-flop egyszerűsített ábrája

Ennek elvégzéséhez a következő lépéseket kell elvégezni:

- az összes, a tervbe behuzalozott flip-flop állapotának beolvasása
- a flip-flopok és a hozzájuk tartozó kapcsolók állapotainak az összehasonlítása
- az FPGA újrakonfigurálása azért hogy megváltoztassuk a kapcsolók állását ott ahol szükséges, a flip-flop állapotainak függvényében, azért hogy a GSR csak annak a flip-flopnak az állapotát módosítsa amelybe a hibát injektálni szeretnénk
- egy GSR impulzus adása
- az FPGA újrakonfigurálása hogy a megváltoztatott kapcsolókat visszaállítsuk eredeti állapotukba, hogy az eszköz kiindulási állapota ne változzon meg

Amint látható, az eszközről egy visszaolvasás (*readback*) és két újrakonfigurálás szükséges egy *SEU* injektálásához, ami túl sok időt vehet igénybe. Ez az idő csökkenthető parciális újrakonfigurálás és visszaolvasás alkalmazásával. A Virtex FPGA-k belső struktúrája korlátozza a parciális újrakonfigurálás hatását, miután egy bizonyos flip-flop újrakonfigurálásához vagy visszaolvasásához az abban az oslzopban lévő ugyanolyan tìpusú (pozíciójú) összes többi flip-flop újrakonfigurálása vagy visszaolvasása szükséges.

A JBits interfész

A JBits API (*Application Programming Interface*) egy olyan Java-alapú interfész ami lehetővé teszi a tervező részére a közvetlen kommunikációt, adatcserét a Xilinx FPGA-val, tehát hogy közvetlenül írjunk ki az eszközre vagy olvassunk be onnan adatokat [59]. A JBits segítségével a *bitstream* gyors módosítására és az eszköz gyors újrakonfigurálására is lehetőség van. Virtex eszközök esetén a JBits lehetőséget nyújt parciális újrakonfigurálásra is. A Virtex architektúrája megőrzi állapotát az újrakonfigurálási fázis alatt.

A JBits applikációk a Java interfészt *Boundary Scanre* is használhatják platformtól függetlenül, olyan eszközök újrakonfigurálására amelyek akár közvetlenül kapcsolódnak gépünkhöz vagy pedig távolról, az Interneten keresztül.

A JBits legfőbb jellemzői

A JBits célja hogy biztosítson egy gyors és egyszerű interfészt a Xilinx Virtex (és régebben a 4000-es) FPGA-k felé [144]. Az alapötlet az volt, hogy támogassuk azokat az applikációkat amelyeknek az FPGA gyors és dinamikus újrakonfigurálására van szüksége. Hagyományos tervezőrendszerekkel egy *bitstream* generálása VHDL leírásból perceket vagy akár órákat is igénybe vehet, pedig bizonyos esetekben akár pár másodperc is megengedhetetlen lehet az alkalmazás időkorlátai miatt.

A JBits több okból is egyedüli abból a szempontból hogy lehetővé teszi olyan problémák kezelését amelyeket a többi fejlesztőrendszer segítségével nem tudnánk megoldani [144]. A legfontosabb tulajdonságok a következők:

- Másodpercenként 10,000 sor fordítása (Microsoft J++ Java fordító)
- *Run-time*, parciális újrakonfigurálás támogatása
- áramkörök futás közbeni parametrizálásának lehetősége
- előre elkészített, behuzalozott makrók (*core*-ok)
- a CLB-ben lévő LUT-ok egyszerű programozása
- hozzáférés minden erőforráshoz
- Virtex család támogatása (kezdeti, ma már nem használt verzió az XC4000EX és XC4000XL családot is támogatta)

- Javaban implementált környezet

Miután a JBits lehetővé teszi FPGA konfigurációk gyors és dinamikus létrehozását, új perspektívák nyílnak meg a már meglévő fejlesztőrendszerekkel kapcsolatban:

Specifikus tervezőrendszerek: lehetőség nyílik olyan applikáció-specifikus fejlesztőrendszerek létrehozására amelyek RTR-t alkalmaznak. Ezek a specifikus rendszerek lehetnek olyan kis méretűek és gyorsak hogy az FPGA-t elfedjék a felhasználó elől. Emellett, mivel a JBits-et Javaban írták, egyéb Java könyvtárak is felhasználhatóak pl. *GUI*-k létrehozásához

Hagyományos fejlesztőrendszerek: mivel a JBits egy programozási interfész a Xilinx FPGA eszközök felé, a JBits API-t használhatjuk huzalozási algoritmusok kifejlesztésére is.

Makrók generálása: előre elkészített makrókat (*core*-ok) lehet létrehozni a JBits-szel. Mivel ezeket a makrókat Javaban írják, parametrizálhatóak és a tervezés folyamán többször felhasználhatóak pl. újabb makrók létrehozásához. Emellett a JBits makrók futási időben parametrizálhatóak és inicializálhatóak. Ez sokkal rugalmasabb tervezést tesz lehetővé mint a statikus, fordítási időben parametrizálható makrók alkalmazása.

Újrakonfigurálás: a terveket dinamikusan lehet változtatni és létrehozni: lehetőség van a különböző paraméterek dinamikus megváltoztatására valamint a leírások dinamikus létrehozására. A JBits olyan flexibilitást tesz lehetővé, amely más fejlesztőrendszereknél nem lehetséges.

A dolgozatban bemutatott munka a JBits környezet első alkalmazása hibainjektálásra. A JBits-ben kifejlesztett program lehetővé teszi az előzőekben bemutatott hibainjektálási módszerek megvalósítását leragadási és *SEU* hibák injektálására.

Eredmények: számítások és mérések

A kifejlesztett módszer esetén egy hibainjektálási kísérlethez szükséges idő 3 részre bontható: az FPGA eszköz inicializálásának ideje (t_{init}), futtatás ideje (t_{fut}) valamint a hibainjektálás ideje (t_{inj}). Az inicializálás ideje az eszköz inicializálását, vagyis a teljes bitstream letöltésének idejét foglalja magába. A futtatási ido az eszköz órajelének léptetésének (vagyis a letöltött applikáció futtatásának) idejét jelenti. Az injektálás ideje a hibának az FPGA eszköz (parciális) újrakonfigurálásával történo injektálásához szükséges időt jelenti. A következő fejezet ezen időket írja le numerikus értékekkel

leragadási hibák és *SEU*-k esetén.

Számítási eredmények

Leragadási hibák injektálására vonatkozó számítások

A számításokat egy egyszerű példa esetén fogjuk végezni: a leragadási hiba injektálását egy XCV50-es eszközön végezzük 1 MHz-es konfigurálási és visszaolvasási (*readback*) frekvencia esetén. A kísérletek során injektálandó hibák száma legyen 10,000 ($N_{hibk} = 10000$) valamint egy kísérletben a futtatási ciklusok száma legyen 1,000 ($N_{ciklus/ksrlet} = 1000$). Az FPGA eszköz konfigurálása és visszaolvasása 8 biten párhuzamosan történik. Nézzük a hibainjektálás klasszikus esetét: egy kísérlet során egy hibát injektálunk és végeznünk kell egy referencia-kísérletet injektált hiba nélkül is. Ennek megfelelően az összes végrehajtandó ciklus száma egyszerűen számolható:

$N_{sszciklus} = (N_{hibk} + 1) * N_{ciklus/ksrlet}$

Az eszköz inicializálási ideje (a teljes bitstream letöltése az FPGA-ba) könnyen kiszámítható, tudva hogy egy XCV50-es eszköz bitstream-je 559200 bitet tartalmaz:

$t_{init} = \frac{559200}{10^3 * 8} = 69.9ms$

Más eszközök inicializálási idői hasonlóképpen számolhatóak.

Egy 0-ba vagy 1-be ragadási hiba injektálásához (és eltávolításához) egy CLB LUT-jának 8 bitjének kétszeri újrakonfigurálására van szükség, amennyiben parciális újrakonfigurálást alkalmazunk. Egy Xilinx Virtex eszköz parciális újrakonfigurálása keretenként (*frame*-ek) történik, mivel egy *frame* a legkisebb újrakonfigurálható egység. Egy *frame* egy bizonyos LUT-nak csak egy bitjét tartalmazza, pl. a 0. számú bitet az összes 0. *slice*-ban található F LUT-ból az adott oszlop esetén. Ennek megfelelően egy LUT 8 bitjének újrakonfigurálásához (vagy visszaolvasásához) 8 *frame* újrakonfigurálása (vagy visszaolvasása) szükséges. Ennek megfelelően egy 0-ba vagy 1-be ragadási hiba injektálásához (és eltávolításához) 8 bit kétszeri újrakonfigurálására van szükség.

Egy XCV50-es eszköz *frame*-je 384 bitet tartalmaz. Ennek megfelelően egy hiba injektálásához (és eltávolításához) szükséges idő a következőképpen számolható:

$t_{egyhibainjektlsa} = \frac{16 * 364}{10^3 * 8} = 7.28 * 10^{-1} ms$

Mivel 10,000 hiba injektálása történik a kísérletsorozat folyamán, a teljes hibainjektálási idő

$t_{inj} = 10000 * t_{egyhibainjektlsa} = 7280ms$

szerint adódik.

Amint fentebb említettük, összesen 10,001 kísérletet végzünk és egy kísérlet során 1000 ciklust kell végrehajtani. A teljes ciklusszám tehát:

$N_{ciklus} = (10000 + 1) * 1000 = 10001000$

A futtatási idő a futási frekvencia (1MHz) ismeretében adódik:

$t_{fut} = \frac{10001000}{10^{-3}} = 10001ms$

A fenti egyenletekből az adott hibainjektálási kísérletsorozat teljes időtartama

$t_{teljes} = 69.9 + 7280 + 100010 = 17,350.9ms$

szerint alakul.

A fentiekhez hasonlóan számolható a hibainjektálási kísérletsorozat időtartama más feltételek, paraméterek mellett (pl. más konfigurálási frekvencia, injektálandó hibaszám stb.).

SEU injektálására vonatkozó számítások

A leragadási hibákhoz hasonlóan a *SEU*-k injektálásához szükséges idő is kiszámolható. *SEU*-k injektálása esetén a flip-flop-ok és hozzájuk tartozó kapcsolók (amelyek a *set*-et és a *reset*-et szabályozzák) egy részének állapotának beolvasása és újrakonfigurálása szükséges. Legrosszabb esetben, annak függvényében hogy az áramköri tervben lévő flip=flopok hogyan lettek elhelyezve az FPGA eszközben, az összes flip-flop és kapcsoló beolvasása/újrakonfigurálása szükséges. Látható, hogy ez a művelet jóval nagyobb mennyiségű adat beolvasását és újrakonfigurálását teszi szükségessé, mint amennyit leragadási hibák esetén szükséges. Legrosszabb esetben (*worst-case*) az összes flip-flop állapotát vissza kell olvasni egyszer és az összes kapcsolót újra kell konfigurálni kétszer minden egyes hiba injektálásához. Egy oszlop összes, megegyező fajtájú flip-flopjainak állapotainak visszaolvasásához két *frame* beolvasása valamint egy 28 byte-os visszaolvasási (*readback*) utasítás kiküldése szükséges ami összesen 992 bit. Miután 4 fajta flip-flop van egy oszlopban és 24 oszlop egy XCV50-es eszközben, a bitek száma ami ahhoz szükséges hogy az egész eszköz összes flip-flop-jának állapotát visszaolvassuk, a következőképpen alakul:

$n_1 = 992 * 4 * 24 = 95232$

Az összes kapcsoló újrakonfigurálásához 11300 byte újrakonfigurálása szükséges. Miután ezt kétszer kell végrehajtani, a művelet során újrakonfigurálandó bitek száma (amennyiben az összes kapcsolót újra kell konfigurálnunk):

$n_2 = 11300 * 2 * 8 = 180800$

A számításoknál ugyanazokat a paramétereket vegyük figyelembe mint amit a leragadási hibák injektálása esetén: 1 MHz-es konfigurálási és visszaolvasási frekvencia, XCV50-es eszköz valamint 8 bit párhuzamos konfigurálása/visszaolvasása. Először *worst-case* esetben számoljuk ki a hibainjektálási kísérlethez szükséges időt, vagyis az összes flip-flop állapotot be kell olvasni és az összes kapcsolót kétszer újra kell konfigurálni minden egyes injektálási lépésben. Ekkor egy *SEU* injektáláshoz szükséges időt a következő egyenlet szerint számolhatjuk:

$t_{injektseu} = \frac{N_{hibk}(n_1+n_2)}{f_{konfvisszaolv}N_{prhbitekszma}}$

Az adott numerikus értékeket a paraméterek helyére behelyettesítve adódik a *worst-case* hibainjektálási idő:

$t_{injektseu} = \frac{(95232+180800)}{10^3*8} = 34.504ms$

A 4 táblázat mutatja több hiba injektálásához szükséges időket *worst-case* esetben.

	10,000 injektált hiba és 1,000 ciklus egy kísérletben	**10,000 injektált hiba és 10,000 ciklus egy kísérletben**	**1,000,000 injektált hiba és 100,000 ciklus egy kísérletben**
Inicializálás (ms)	69.9	69.9	69.9
Futtatási ido (min)	0.167	1.67	1,666.67
Injektálási idő (min)	5.75	5.75	575.07
Teljes idő (min)	5.92	7.42	2,241.74

4. táblázat.
Hibainjektálási kísérletsorozathoz szükséges idők *SEU*-k injektálásánál *worst-case* esetben, parciális újrakonfigurálást alkalmazásával

Amint azt már említettük, ezen számításokat a legrosszabb eset (*worst-case*) figyelembe vételével végeztük. Jobb esetben azonban előfordulhat pl. hogy a kapcsolóknak/flip-flop-oknak csak egy részét kell újrakonfigurálni/visszaolvasni, a szintézis után futtatott huzalozási algoritmustól és a tervnek az FPGA-ban elfoglalt helyének nagyságától függően. A fentebb feltételezett legrosszabb esetben az eszköz 100%-át kitölti a terv, ráadásul igen alacsony konfigurálási frekvenciát (1 MHz) tételeztünk fel.

Vegyünk most egy ennél kedvezőbb esetet: az eszköznek csak a 75%-át tölti ki a szintetizált terv (ennek megfelelően a flip-flop-oknak és kapcsolóknak csak 75%-át kell visszaolvasnunk/újrakonfigurálnunk), valamint a konfigurálási/visszaolvasási frekvencia 10 MHz (a többi paraméter változatlan).

A flip-flop-ok állapotainak visszaolvasásához szükséges bitek száma ekkor

$n_1 = 95232 * 0.75 = 71424$

szerint alakul.

Hasonlóképpen számolható az előzőekből a kapcsolók újrakonfigurálásához szükséges bitek száma:

$n_2 = 180800 * 0.75 = 135600$

A fentiekbol adódik egy *SEU* injektálásához szükséges ido (hasonlóképpen számítva mint *worst-case* esetben):

$t_{injektseu} = \frac{(71424+135600)}{10^4*8} = 2.587ms$

A hibainjektálási kísérletsorozathoz szükséges időket a 5 táblázat foglalja össze.

	10,000 injektált hiba és 1,000 ciklus egy kísérletben	**10,000 injektált hiba és 10,000 ciklus egy kísérletben**	**1,000,000 injektált hiba és 100,000 ciklus egy kísérletben**
Inicializálás (ms)	69.9	69.9	69.9
Futtatási ido (min)	0.167	1.67	1,666.67
Injektálási ido (min)	0.43	0.43	43.12
Teljes ido (min)	0.60	2.10	1,709.79

5. táblázat.
SEU-k injektálásához szükséges idők kedvezőbb feltételek között parciális újrakonfigurálás alkalmazása esetén

Ha összehasonlítjuk a 4 és 5 táblázatokat, láthatjuk, hogy 1 nagyságrendű időbeli nyereség érhető el ha az adott feltételek nem *worst-case* hanem annál valamivel kedvezőbb (de nem optimális) feltételek.

A hibainjektálási kísérletsorozathoz szükséges időnek a különböző paraméterektől való függésének a vizsgálata során az újrakonfigurálandó bitek hatását is tanumányoztuk. Ennek alapjń kijelenthetjük, hogy a Virtex FPGA család (jelenlegi) architektúrája nem a legkedvezöbb ilyen fajta kísérletek végrehajtására.

A dolgozatban bemutatott új hibainjektálási technikának a már korábban ismert (szimuláción- illetve emuláción alapuló) módszerekkel való összehasonlítása azt mutatja, hogy a bemutatott technika az időtartam szempontjából jóval hatékonyabb mint a szimuláció, és különböző esetekben versenyképes a klasszikus emulációval. A módszer hatékonysága erősen függ az injektált hibḱ számától (jobban mint a ciklusok számától) klasszikus emulációval történő összehasonlítás esetén.

Mérési eredmények

A kísérletek egy XSV50-es demo kártyán lettek elvégezve [142] a módszer megvalósíthatóságának bemutatása céljából.. Ezt a kártyát valamint a kártyához írt, JBits-hez

szükséges Java interfészt nem magas frekvenciát igénylő kísérletek végrehajtásához tervezték; a kártyán lévő FPGA eszköz újrakonfigurálása és az eszközről az adatok visszaolvasása az alacsony átviteli sebességű (2Mbs) párhuzamos porton keresztül történik. A kártya konfigurálási/visszaolvasási sebessége a JBits és az XHWIF interfészen segítségével mindössze 3.9 kHz. Ilyen feltételek mellett az előzőekben bemutatott számítási eredmények kimérése nem lehetséges. A bemutatott eredmények gyakorlatban való eléréséhez és így a dolgozatban leírt módszer által nyújtotta lehetoségek kihasználásához egy applikáció (azaz hibainjektálás)-specifikus környezetre lenne szükség.

A kísérletek mérési eredményeit *SEU* injektálása esetén tartalmazza a 6 táblázat. Összehasonlítva a 6 és a 5 táblázatokat, látható hogy a mért értékek jóval magasabbak az előzőekben számolt értékeknél a kártya sebességbeli korlátai miatt.

Inicializálás (ms)	**Hibainjektálás (ms)**	**Futtatás (ms)**
19,678	3,566	55,961
19,688	3,575	55,873
19,718	3,498	55,912

6. táblázat.
SEU injektálási kísérletek mérési eredményei XSV50 kártyán (3 kísérlet)

Konklúzió, perspektívák

A doktori kutatás tárgya egy új hibainjektálási technika létrehozása volt hardver prototípus és parciális újrakonfigurálás alkalmazásával. A módszer kétfajta hibára lett kidolgozva és megvalósítva: leragadási hibára és *SEU*-re. A módszer megvalósíthatósága kísérletekben lett megmutatva. A kísérleteket egy Xilinx Virtex FPGA eszközön végeztük a JBits interfész segítségével, ami lehetővé teszi az eszköz parciális újrakonfigurálását.

A számítások és mérések azt mutatják, hogy egy leragadási hiba injektálása milliszekundumos, míg egy *SEU*-é néhány milliszekundumos nagyságrendű.

Egy *SEU* injektálási kampány a dolgozatban bemutatott technikát alkalmazva nem feltétlenül lesz gyorsabb mint a hagyományos hibainjektálási technikákkal (klasszikus hardver prototípus alkalmazás) végezve. Az eredmények azt mutatják hogy a bemutatott módszerrel végzett kísérletekhez szükséges idő több tényezőtől is függ,

amelyek optimalizálása fontos lenne. Hatékony hibainjektálási kísérletek megvalósításához optimalizált környezet szükségeltetne. Ennek a környezetnek lehetővé kellene tennie az FPGA-hoz való gyors hozzáférést, hogy ki tudjuk használni az eszköz technológiai adottságait (gyors újrakonfigurálás/visszaolvasás). Az áramköri le'irásnak az FPGA eszközbe való leképzését is optimalizálni kellene, hogy az újrakonfigurálandó *frame*-ek számát minimálisra lehessen csökkenteni. Esetleg más FPGA családok (pl. AT6K, AT40K [11] [14]) architektúrája kedvezőbb lehet a módszer implementálásához.

Egy ilyen hibainjektálásra-optimalizált környezet megvalósítása perspektívát jelenthet a munka folytatásához.

Chapter 1

Introduction

The principle of fault injection has been known for a long time to evaluate the dependability of a given hardware, software or system implementation. The basic idea is to deliberately create faults into the environment under test after putting it into operation. The system under test is excited with application test vectors and data are collected on the outputs and also potentially on internal signals. At the end, these data can be used in order to analyse the behaviour of the system when faults occur.

Fault injection can be hardware- or software-based. In hardware-based techniques it was proposed to take advantage of hardware prototyping to improve and accelerate the execution of the whole fault injection campaign. The execution of an application can indeed be much faster in hardware than in simulation, so important time gains can be achieved by using prototyping-based techniques. In consequence, the dependability analysis of a circuit can be more extensive on a prototype than in time-consuming simulations.

Reconfigurable hardware (and especially FPGA devices) is a good candidate to implement prototypes and perform fault injections. The reconfiguration of an FPGA can however take a long time and this can be a limitation of prototyping-based techniques, especially if the device must be reconfigured several times. To overcome this problem, partial (also called local) reconfiguration of the hardware was proposed. In this case, only a part of the device must be reconfigured when changes are made. The use of partial reconfiguration capabilities results in an important time gain when only few differences exist between two successive configurations of the FPGA.

This work focuses on fault injections performed in reconfigurable hardware. In particular, it is proposed to take advantage of partial reconfiguration capabilities to perform fault injections in the prototype of a digital circuit. Till now, hardware

prototyping was used for the execution of the application on faulty versions of the circuit. The fault injection itself was generally made by means of internal logic elements controlled by external signals. These elements were added by modifying either the high level description (e.g. behavioural VHDL) or the gate level description of the circuit, before implementing the prototype. The idea developed in this thesis is not only to execute the application in reconfigurable hardware but also to realise the injection of faults directly in the device (FPGA), taking advantage of the reconfiguration capabilities. Like this, each fault injection (or fault removal) necessitates a partial reconfiguration of the device. On the other hand, the initial description of the system must not be changed before implementing the prototype. The main goals of this thesis are to demonstrate the feasibility of such an approach, to show how it can be automated in an up-to-date design flow, to analyse the advantages and limitations with respect to existing techniques and finally to identify the major parameters that must be optimised to implement an efficient fault injection system based on partial reconfiguration.

Chapter 2 gives a brief overview of reconfigurable computing, with emphasis on partial reconfiguration and the environment used in the PhD work. The state of the art of fault injection techniques is discussed in chapter 3. The description of the developed new fault injection methodology can be found in chapter 4. Calculated and measured results are presented in chapter 5 and some comparisons are made with previous techniques. Finally, a conclusion is given in chapter 6.

Chapter 2

Reconfigurable hardware

In this chapter reconfigurable computing is analysed. The "history" of configurable hardware is briefly discussed, and partial reconfiguration is detailed with emphasis.

2.1 General description, comparisons

This section contains the description of reconfigurable hardware, presenting the different kinds of programmable logic. The history of reconfigurable computing is briefly described, and a comparison is made between programmable and fixed logic. Section 2.1.1 contains a general description of reconfigurable computing. A comparison of programmable logic and ASICs is made in section 2.1.2.

2.1.1 Description and brief history of reconfigurable computing

In the world of digital electronic systems, there are three basic kinds of devices: memory, microprocessors, and logic. Memory devices store random information such as the contents of a spreadsheet or database. Microprocessors execute software instructions to perform a wide variety of tasks such as running a word processing program, an IC design tool or a simple game. Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.

Logic devices can be classified into two broad categories - fixed and programmable. As the name suggests, the circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed. On

the other hand, programmable logic devices (PLDs) are standard, off-the-shelf parts that offer customers a wide range of logic capacity, features, speed, and voltage characteristics - and these devices can be changed at any time to perform any number of functions [148].

It was about 1975, when standard fixed logic devices were the rage and printed circuit boards were loaded with them, that the question of reprogrammability has arisen: it would be desired to have the ability during design to implement different interconnections in a large device? Like this, the designer would be able to integrate many standard logic devices into one part. It was Ron Cline from Signetics (which was later purchased by Philips and then Xilinx) who came up with the idea of two programmable planes which provided any combination of AND and OR gates and sharing of AND terms across multiple ORs [109]. These were the first steps in the field of programmable logic and the basis of the first PLDs (PAL, PLA, PROM, GAL, see section 2.2).

2.1.2 Comparison of programmable logic and fixed logic

Programmable and fixed logic can be compared from many aspects concluding that both techniques possess advantages and disadvantages. Like this, it cannot be determined explicitly which technique is "better", a comparison can be made from different points of view and it depends on the conditions and goal of the application which technique fits better. In the following, a brief comparison of the two techniques is made from different aspects.

With fixed logic devices, the time required to go from design, to prototypes, to a final manufacturing run can take from several months to more than a year, depending on the complexity of the device. And, if the device does not work properly, or if the requirements change, a new design must be developed. The up-front work of designing and verifying fixed logic devices involves substantial "non-recurring engineering" (NRE) costs. NRE represents all the costs customers incur before the final fixed logic device emerges from a silicon foundry, including engineering resources, expensive software design tools, expensive photolithography mask sets for manufacturing the various layers of the chip, and the cost of initial prototype devices. These NRE costs can run from a few hundred thousand to several million dollars.

With programmable logic devices, designers use less expensive software tools to develop, simulate, and test their designs. Then, a design can be quickly programmed

into a device, and immediately tested in a live circuit. The PLD that is used for this prototyping is the exact same PLD that will be used in the final production of a piece of end equipment. There are smaller NRE costs and the final design is completed much faster than that of a custom, fixed logic device [58].

Another key benefit of using PLDs is that during the prototyping phase customers can change the circuitry as often as they want until the design operates to their satisfaction. That is because most PLDs can be based on re-writable memory technology - to change the design, the device is simply reprogrammed. Once the design is final, customers can go into immediate production by simply programming as many PLDs as they need with the final software design file.

Of course, reconfigurable logic represents some inconveniences, too. One basic disadvantage of programmable logic is the high unit cost: the fabrication of reconfigurable hardware is much more expensive than the one of ASICs. This inconvenience follows from another disadvantage of programmable logic: the high complexity of architecture. Furthermore, the complexity of the circuitry introduces another inconvenience: the electronic performance of programmable logic is worse than the one of fixed logic, the operating speed, frequency is lower because of the important delays in the circuit. Moreover, higher complexity results in higher sensibility to external conditions.

The advantages and disadvantages of programmable logics, comparisons of configurable and fixed logic are shown on Table 2.1.

	Programmable logic	**Fixed logic**
Advantage	Fast time to market, off the shelf delivery, small NRE costs, possibility of reconfiguration: changes can be made during prototyping phase	Relatively low unit cost, good performance in speed, low sensibility, relatively simple architecture
Disadvantage	High unit cost, low speed, high complexity of architecture, relatively high sensibility	No flexibility, NRE costs, large time to market

Table 2.1: Comparison of programmable logic and fixed logic

2.2 The different kinds of configurable logics

In this section a brief classification of the PLDs is given. This section is divided into two major parts: section 2.2.1 contains a description of the early PLDs called Simple PLDs (SPLD), while the more complex PLDs (Field Programmable Logic Device - FPLD) are reviewed in section 2.2.2.

2.2.1 The SPLDs

It was already mentioned in section 2.1.1 that the idea of reconfigurable computing was born in the 70's. These preliminary PLD families (also called Simple PLDs - SPLD) may contain two programmable interconnection matrix, an "AND" matrix and an "OR" matrix, with which many combination of AND and OR gates (it is the number of AND gates that determines the maximal number of terms), like this many logic functions of the same number of inputs as the PLD device can be implemented. The connection in the intersections of the nets can be established/cleared by fuses: with the aid of software tools, the user can select which junctions will not be connected by "blowing" all unwanted fuses [109]. The simplified schema of this kind of SPLD is shown on fig. 2.1.

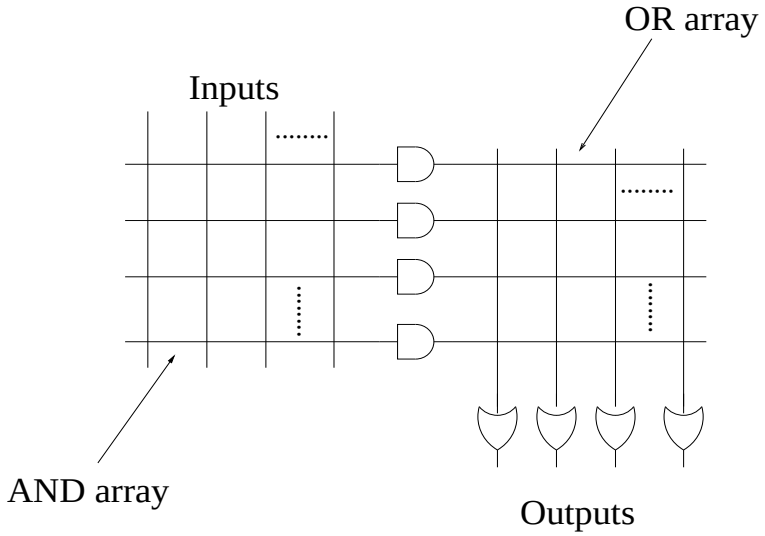


Figure 2.1: The architecture of an SPLD

Based on this architecture, different kinds of PLDs can be fabricated in function of the programmability of the "AND" and "OR" arrays. In the basic case, when both matrix are programmable, it is called Programmable Logic Array (PLA). From this

kind of architecture any combination of AND and OR gates can be derived (until the number of AND gates) and it assures the highest logic density. This results in a disadvantage: the high fuse count. It also has a high propagation delay.

To get around these problems, a slightly different architecture was designed called Programmable Array Logic (PAL). In case of a PAL, the "OR" array is fixed, like this the propagation delay is lower and less complex software is needed to program it as the logic density is lower.

A similar architecture to PALs called Programmable Read Only Memory (PROM) can be obtained in a similar way: the "AND" array of the device is fixed in this case and the "OR" array is programmable, in the same way as described above. On the other hand, it assures that all combinations of AND and OR gates can be implemented (the number of AND gates assures the implementation of all possible minterms).

The fuse-technology of SPLDs has evaluated later: in case of flash PLDs, for example, the device can be programmed and erased electrically.

Another kind of SPLD is the family of GALs (Generic Array Logic). A GAL device is very similar to the PAL structure: the "OR" array of the device is fixed and the "AND" array is programmable. However, in case of GALs, the outputs of the "OR" matrix are connected to storage elements (flip-flops), like this not only combinational logics (as in case of the other kinds of SPLDs) but sequential ones can be implemented in GALs.

2.2.2 The FPLDs

With the growing necessity to more complex logic designs, the extension of the density of the simple PLDs was needed. Like this, more complex programmable logics were created to satisfy this need. These devices are called Field Programmable Logic Devices (FPLD). Two major types of FPLDs can be distinguished (and are described in this section): the Complex Programmable Logic Devices (CPLDs) and the Field Programmable Gate Arrays (FPGAs).

The CPLDs

CPLDs are similar to SPLDs except that they have significantly higher capacity. A typical CPLD is the equivalent of two to 64 SPLDs. A CPLD typically contains from tens to a few hundred cells (macrocells) which are PLD-like blocks and connected to each-other (fig. 2.2).

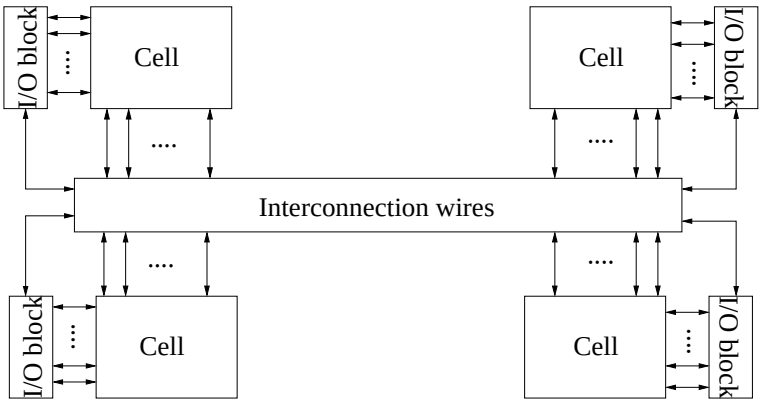


Figure 2.2: Part of the architecture of a CPLD

A macrocell on most modern CPLDs contains a sum-of-products combinatorial logic function and one or several optional flip-flops. The combinatorial logic function typically supports four to sixteen product terms with wide fan-in. In other words, a macrocell may have many inputs, but the complexity of the logic function is limited. Fig. 2.3 shows a possible macrocell architecture.

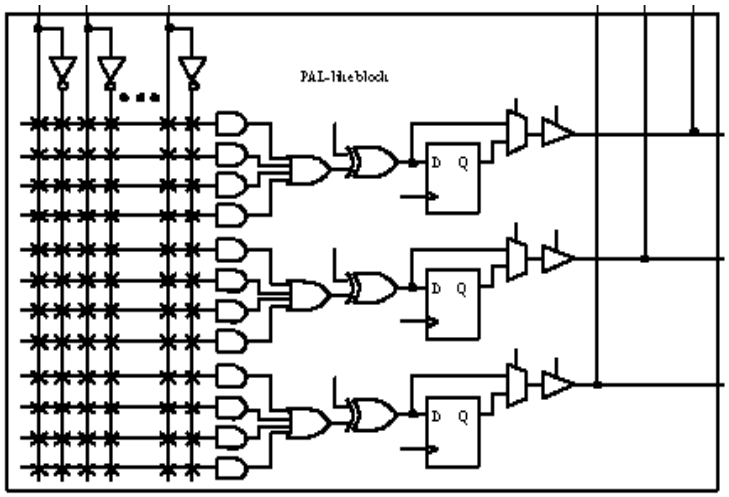


Figure 2.3: The basic architecture of a macrocell

Sets of eight to 16 macrocells are typically grouped together into a larger function block. The macrocells within a function block are usually fully connected. If a device contains multiple function blocks, then the function blocks are further interconnected. Not all CPLDs are fully connected between function blocks - this is vendor and family specific. Less than 100% connection can in general be used between function blocks, that means that there is a chance the device will not route or may have problems

keeping the same pinout between design revisions [108].

CPLDs provide a natural migration path for SPLD designers seeking higher density, as CPLDs have a PAL-like architecture and generally four or more PALs comfortably fit into a CPLD. Most CPLDs support the SPLD development languages such as ABEL, PALASM, CUPL etc.

CPLDs offer very predictable timing characteristics and fast pin-to-pin performance, like this they are great at handling wide and complex gating at blistering speeds, e.g. 5ns which is equivalent to 200MHz. The timing model for CPLDs is easy to calculate, so even before starting the design it is possible to calculate the input to output speeds. Like this, CPLDs are generally best for control-oriented designs: the wide fan-in of their macrocells makes them well-suited to complex, high performance state machines.

Some of the major variations between CPLD architectures include the number of product terms per macrocell, whether product terms from one macrocell can be borrowed or allocated to another macrocell, and whether the interconnect switch matrix is fully- or partially-populated.

In some architectures, when the required number of product terms exceeds the number available in the macrocell, additional product terms are borrowed from an adjoining macrocell. This makes the CPLD device useful for a wider variety of applications. When borrowing product terms from an adjoining macrocell, that macrocell may no longer be useful. In some architectures, the macrocell still has some basic functionality. Borrowed product terms usually means increased propagation delay.

An important feature of the CPLDs is the number of connections within the switch matrix. A switch matrix supporting all possible connections is fully populated. A partially- populated switch supports most, but not all, connections. The number of connections within the switch matrix determines how easy a design will fit in a given device. With a fully-populated switch matrix, a design will route even with a majority of the device resources used and with fixed I/O pin assignment. Generally, the delays within a fully populated switch matrix are fixed and predictable [108].

A device with a partially-populated switch matrix may have problems routing complex designs. Also, it may be difficult to make design changes in these devices without using a different pinout. Routing to a fixed pinout is important. It is far easier to change the internals of a programmable logic device than it is to re-layout a circuit board. The delays within a partially-populated switch matrix are not fixed and

less easily predicted, similar to most FPGA devices. Though a partially populated switch matrix has some potential limitations, it is less expensive to manufacture.

CPLDs are manufactured using one of three process technologies-EPROM, EEPROM, or FLASH. EPROM-based CPLDs are usually one-time programmable (OTP) unless they are in an UV-erasable windowed package. A device programmer or the manufacturer or distributor programs an EPROM-based CPLD.

Generally, CPLDs are CMOS and use non-volatile memory cells such as EPROM, EEPROM, or FLASH to define the functionality. Many of the most-recently introduced CPLD families use a EEPROM or FLASH and have been designed so that they can be programmed in-circuit (also called ISP for in-system programmable) [108].

The FPGAs

As it is mentioned at the beginning of section 2.2.2, the other major type of FPLDs (besides CPLDs) are the FPGAs. It was in 1985 that Xilinx introduced this new idea. The basic concept was to combine the user control and time to market of PLDs with the densities and cost benefits of gate arrays.

FPGAs are distinct from SPLDs and CPLDs and typically offer the highest logic capacity. An FPGA consists of an array of logic blocks (called configurable Logic Blocks - CLB), surrounded by programmable I/O blocks, and connected with programmable interconnect. Fig. 2.4 shows the basic structure of an FPGA.

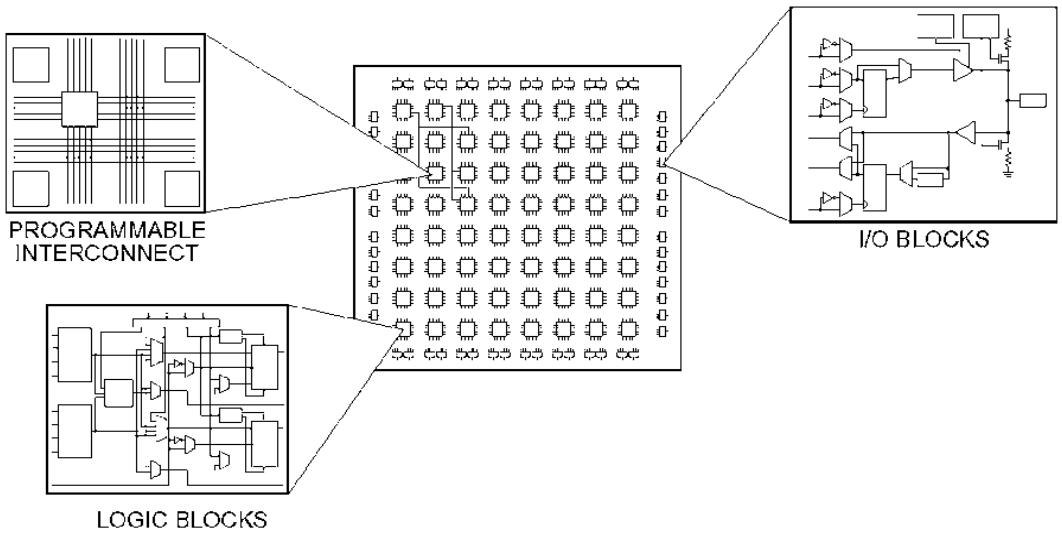


Figure 2.4: Simplified schema of an FPGA architecture

A typical FPGA contains from 64 to tens of thousands of CLBs and an even

greater number of flip-flops, as a CLB contains generally more than one flip-flop. Most FPGAs do not provide 100% interconnect between CLBs (to do so would be prohibitively expensive). Instead, sophisticated software places and routes the logic on the device much like a PCB autorouter would place and route components.

When compared to CPLDs, FPGAs offer a higher amount of logic density, more features and higher performance. The largest FPGA now available (part of the Virtex family) provides eight million of "system gates" (the relative density of logic). The secret to density and performance in these devices lies in the logic contained in their CLBs and on the performance and efficiency of their routing architecture.

FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing. These advanced devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies. Such a revolutionary product that combines the processing power of a microcontroller with the flexibility of an FPGA is the Field Programmable System Level Integrated Circuit (FPSLIC) family of Atmel (AT94K [12] and AT94S [13]). This integrates, on a single IC, an advanced 8-bit RISC microcontroller (AVR) core, peripherals and SRAM program memory, together with a large FPGA block. A secured variant encapsulates, in a single package, an FPSLIC IC together with its EEPROM for application and configuration code. This reduces board size and makes the code resistant to reverse engineering. Co-verification tools are also available to enable concurrent hardware and software development for FPSLIC applications.

FPGAs can be classified in different ways. One possible way is to consider the granularity of the architecture. From this viewpoint, two primary classes of FPGA architectures can be identified:

- coarse-grained
- fine-grained

Coarse-grained architectures consist of fairly large logic blocks, often containing two or more look-up tables and two or more flip-flops. In a majority of these architectures, a four-input look-up table (think of it as a 16x1 ROM) implements the actual logic. The larger logic block usually corresponds to improved performance.

In devices of fine-grained architecture there are a large number of relatively simple logic blocks. The logic block usually contains either a two-input logic function or a 4-to-1 multiplexer and a flip-flop. These devices are good at systolic functions and have some benefits for designs created by logic synthesis.

Another way to classify the FPGAs is to consider the underlying process technology used to program the device. Two basic technologies are used currently to build FPGA devices:

- static memory
- anti-fuse

The static memory-based technology is similar to the technology used in static RAM devices but with a few modifications. The RAM cells in a memory device are designed for fastest possible read/write performance. The RAM cells in a programmable device are usually designed for stability instead of read/write performance. Consequently, RAM cells in a programmable device have a low-impedance connect to voltage supply and ground to provide maximum stability over voltage fluctuations.

Because static memory is volatile (the contents disappear when the power is turned off), SRAM-based devices are "booted" after power-on. This makes them in-system programmable and re-programmable, even in real-time. A device is called in-circuit programmable or in-system programmable (ISP) if it can be programmed while it is mounted on the circuit board with the other components. As a result, SRAM-based FPGAs are common in reconfigurable computing applications where the device's function is dynamically changed.

The configuration process typically requires only a few hundred milliseconds at most. Most SRAM-based devices can boot themselves automatically at power-on much like a microprocessor. Furthermore, most SRAM-based devices are designed to work with either standard byte-wide PROMs or with sequential-access serial PROMs. SRAM cells are also used in many non-volatile CPLDs to hold some configuration bits to reduce internal capacitive loading [108].

To reprogram the device, some form of external configuration memory source is required. The configuration memory holds the program that defines how each of the logic blocks functions, which I/O blocks are inputs and outputs, and how the blocks are interconnected together.

Devices of the other class of FPGAs, based on anti-fuse technology are one-time programmable (OTP). Once programmed, they cannot be modified, but they also retain their program when the power is off. Anti-fuse devices are programmed in a device programmer either by the end user or by the factory or distributor.

These devices are called anti-fuse devices because of their programming method. Instead of breaking a metal connection by passing current through it, a link is grown to make a connection. Anti-fuses are either amorphous silicon or metal-to-metal connections. They are usually physically quite small and have low on-resistance. Consequently, anti-fuse technology has benefits for creating programmable interconnect. However, they require large programming transistors on the device [108].

Some FPGAs have built-in system-level features like on-chip bussing, on-chip RAM for building small register files or FIFOs, and built-in JTAG boundary-scan support.

Besides the two most important above described technologies, there is another, not really spread technology to build FPGA devices: using FLASH technology. FLASH memories are electrically erasable programmable devices having the electrically erasable benefits of EEPROM but the small, economical cell size of EPROM technology.

The FPGA manufacturers are regrouped on table 2.2 in function of the type of FPGAs they fabricate [108].

Architecture	**Static Memory**	**Anti-fuse**	**Flash**
Coarse-grained	Altera: FLEX, APEX Atmel: AT40K DynaChip Lucent: ORCA Vantis: VF1 Xilinx: XC3000, XC4000xx, Spartan, Virtex	QuickLogic: pASIC	
Fine-grained	Actel: SPGA Atmel: AT6000	Actel: ACT	Gatefield

Table 2.2: Summary of FPGAs and manufacturers concerning architectures and technology processes

2.3 Reconfiguration techniques

This section deals with the different reconfiguration modes of FPGA devices. The main aspect of classification is whether the application which is executed in the FPGA device is constituted from one or more configuration(s). In the first case, the configuration is loaded into the FPGA device at the beginning so no reconfiguration is needed during the execution of the application (section 2.3.1). On the other hand, the device must be reconfigured during the execution of the application if the application consists of more than one configuration (section 2.3.2).

Different terminologies can be found for this kind of classification of FPGA-based computing. In [85] Kung introduces the terms "compile-time" and "run-time" to qualify the reconfiguration of VLSI processor arrays. Lysaght et al. in [95] apply these terms to FPGAs whether the FPGAs are reconfigured during the execution of the application (run-time) or only at the beginning (compile-time).

In [95] FPGAs are also classified whether it is possible to reconfigure only part of it at a time or the whole device must be reconfigured at each configuration step. The first kind of reconfiguration is called partial reconfiguration. More precisely, they refer in [95] to partial reconfiguration if it is possible to selectively reconfigure the device, while the rest of the device remains inactive but retains its configuration information. Moreover, in [95] the device is called not only partially but dynamically reconfigurable if their embedded configuration storage circuitry can be updated selectively, i. e. nominated configuration storage circuits (and the corresponding logic functions and interconnections which they control) can be changed without disturbing the operation of the remaining logic. It means that a part of the circuit can be reconfigured while the rest of the device remains in operation. In [4] the terms "Active Local RTR" and "Passive Local RTR", where RTR stands for Run-Time Reconfiguration, are used for dynamic reconfiguration and partial reconfiguration.

Hutchings et al. refers also to RTR in [51] [53] [52]. The same terminology (compile-time, or CTR, and run-time reconfiguration) is used in [64], and two kinds of RTR are identified: Global RTR and Local RTR. In the first case (Global RTR), all the device is reconfigured at each configuration step. Local RTR refers to the same operation as partial reconfiguration in [95]. In further publications of the authors [140] [61] this operation is generally referred as partial reconfiguration.

There are some other, less frequently used terms for the above mentioned operations. Johnson refers to dynamic reconfiguration as "real-time reconfiguration"

in [73]. Apel uses the term "on-line reconfiguration" in place of dynamic reconfiguration [7]. Chean et al. uses the term "static reconfiguration" for CTR in case of the reconfiguration of fault-tolerant processor arrays [36].

In the following, the terms of CTR, RTR and partial reconfiguration (in some cases, Local RTR) will be used.

2.3.1 Compile-Time Reconfiguration

CTR is the simplest and most commonly-used approach for implementing applications with reconfigurable logic [64]. The most important feature of CTR applications is that they consist of a single system-wide configuration for all the system (Fig. 2.5). The FPGAs are loaded with their respective configurations before starting the operation, and once execution of the application starts, they remain in this configuration till the end of execution. This approach is similar to using an ASIC because the hardware does not change during the execution of the application.

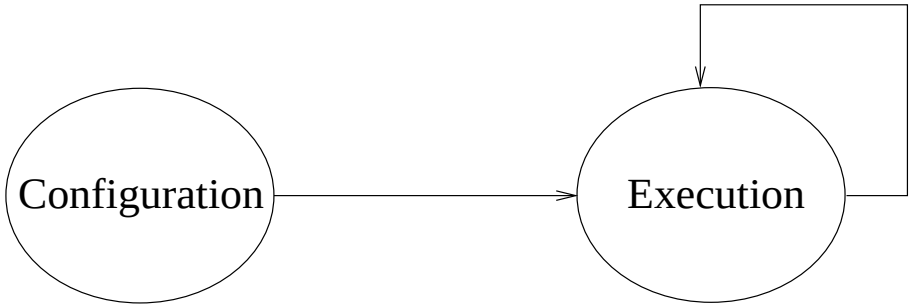


Figure 2.5: Compile-Time Reconfiguration (CTR)

2.3.2 Run-Time Reconfiguration

The technique of Run-Time Reconfiguration (RTR) is to reconfigure hardware resources during application execution. This approach has been proposed for example in [95] [64]. Contrary to CTR, RTR systems reconfigure the FPGA several times during the execution of the application [64].

One can distinguish different kinds of RTR methods. When the whole FPGA is reconfigured during a reconfiguration (Global RTR) with an arbitrary number of executions for each configuration, the execution of the application must be stopped during reconfiguration. In this case the application is divided into distinct temporal phases where each phase is implemented as a single system-wide configuration that

occupies all system FPGA resources [64]. The process of dividing an algorithm into time-exclusive segments is referred to as temporal partitioning. Fig. 2.6 shows the execution of a Global RTR application which is mapped into 2 configurations.

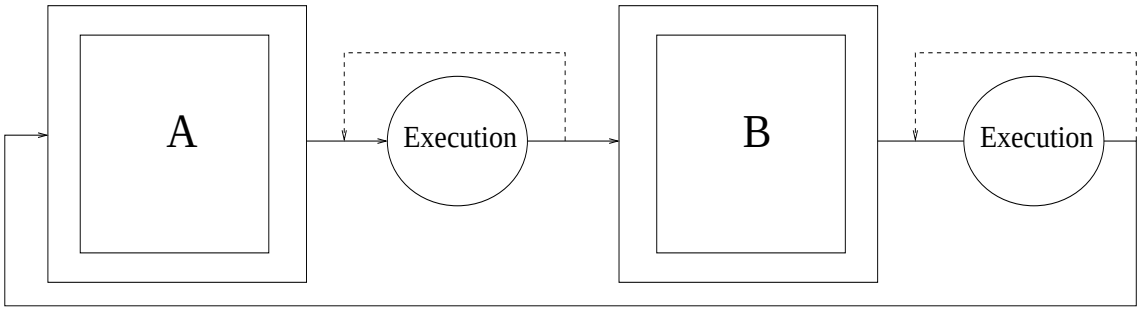


Figure 2.6: Global Run-Time Reconfiguration

The main advantage of Global RTR is its relative simplicity. Because the application partitions are coarse grained, conventional CAD tools (schematic capture, VHDL or Verilog synthesis etc.) can be used successfully once the manual temporal partitioning step has been completed. Each partition can be designed and implemented as one independent circuit module and the CAD tool is free to perform any global optimisations as necessary [64].

It is also possible to reconfigure only subsets of the reconfigurable circuit. This approach is called partial reconfiguration or Local RTR. In this case important time-savings are made compared with a complete reconfiguration of the components, as reconfiguration is quite a time-consuming operation and with Local RTR not all the circuitry must be reconfigured to carry out changes. Fig. 2.7 shows an example of Local RTR where the application to implement consists of 4 partitions: A, B, C and D. In a first step, partitions "A", "B" and "C" are loaded into the FPGA and then executed. In a second step, partitions "B" and "C" are removed and partition "D" is loaded into the FPGA, which is followed by the execution of the second part of the application.

As it is mentioned already at the beginning of section 2.3, local (partial) RTR can be carried out in two different ways: during the reconfiguration of part of the circuit the operation of the other parts is either interrupted or not.

The organization of partial RTR applications is based more on a functional division of labor than the phased partitioning used by Global RTR applications. Local RTR applications are typically implemented by functionally partitioning an application into a set of fine-grained application operations. These operations need not be entirely

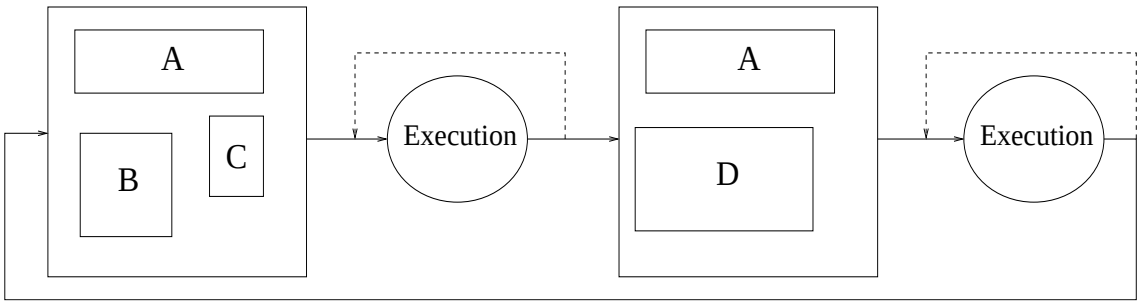


Figure 2.7: Local (partial) Run-Time Reconfiguration

temporally exclusive as many of them may be active at any time. This is in direct contrast to global RTR where only one configuration (per FPGA) may be active at any given time. Still, it is important to organize the operations such that idle circuitry is eliminated or greatly reduced. Each of these operations is implemented as a distinct circuit module and these circuit modules are then downloaded to the FPGAs as necessary during the operation of the application. Note that, unlike global RTR, several of these operations may be loaded simultaneously and each operation may consume any portion of the system FPGA resources [64].

The main advantage that partial RTR provides over Global RTR is the ability to create fine-grained functional operators that make more efficient the use of FPGA resources. This is important for applications that are not easily divided into equal-sized temporally exclusive circuit partitions. However, this advantage can cause a very high design penalty because of the increased flexibility and complexity of the system. Unlike Global RTR where circuit interfaces remain fixed between configurations, local RTR allows these interfaces to change with each configuration. For example, when circuit configurations become small enough for multiple configurations to fit in a single device, the designer is now forced to ensure that all configurations will interface correctly, one with another. Moreover, the designer may also have to ensure not only structural compliance but physical compliance as well. That is, when the designer creates circuit configurations that do not occupy an entire FPGA, they will have to ensure that the physical footprint of a configuration is compatible with other configurations that may currently be loaded. In addition, the designer must ensure that any physical circuitry is placed and routed such that inter-configuration communication is supported properly [64].

Another disadvantage of partial RTR is the very few devices (e.g. the Virtex families of Xilinx [145] or the AT6K and AT40K families of Atmel [11] [14]) and

corresponding tools that are capable to realise the partial reconfiguration of the FPGA device. Traditional CAD tools do not support generally this feature, specific tools like JBits (section 2.4.3) are needed to take advantage of partial reconfiguration.

Implementation examples using partial RTR

The technique of partial reconfiguration is not new at all as it was already mentioned at the beginning of section 2.3.2. When the technique has been proposed, only a few FPGA families supported partial reconfiguration and CAD tools were a very poor match for partial reconfiguration implementations. Nowadays, there are already lots of application examples applying the technique of partial RTR. The description of all of them is not the goal of this PhD thesis, so only the first implementation issues will be mentioned in this section.

Lysaght et al. reported on a design that emulates a large neural network by reconfiguring the network between layers [96].

The implementation of neural networking hardware was also the subject of [51] [53] [52]. In this work, the backpropagation algorithm was implemented by reconfiguring between algorithm stages in order to eliminate idle circuitry and allow additional concurrency.

A general processor architecture has also been proposed in [140] that uses partial RTR to offset the limited resources available on an FPGA and allow an essentially infinite application specific instruction set.

2.3.3 Brief comparison of CTR and RTR techniques

The comparison of programmable logic and fixed logic can be found in section 2.1.2 and section 2.3.2 contains the comparison of Global and Local RTR. This section deals with the comparison of CTR and RTR techniques.

Comparing to CTR applications, the density of the circuitry is enhanced when applying RTR as it has been proven by Wirthlin et al. in [141] where a mathematical formalism is also proposed to describe RTR systems. Applications larger than the FPGA capacity can be run using RTR as only part of the whole application must be present in the device at a time.

On the other hand, the dynamic nature of the hardware introduces two new design problems. The first is dividing the algorithm into time-exclusive segments that do not need to (or cannot) run concurrently (temporal partitioning). Similar to

structural partitioning where circuit elements are organized into strongly connected partitions, temporal partitioning is a process that organizes circuit elements into strongly concurrent partitions, i.e., groups of circuit elements that must or tend to operate concurrently. Typically, an algorithm is temporally partitioned by breaking it down into distinct phases or operational modes. Each of these phases or modes is then designed as a distinct circuit module (FPGA configuration). These configuration are then downloaded into the FPGAs as required by the application [64].

The configurations that make up an RTR application should be organized such that they can remain resident for a reasonable amount of time. In addition, they should perform their task relatively independent of other configurations. Because most digital design tools assume a static hardware model, very few tools support this partitioning step.

The second design problem introduced by RTR systems involves coordinating the behaviour between configurations of RTR applications. This behaviour is manifested in the form of inter-configuration communication, or the transmission of intermediate results from one configuration to the next. This occurs during the normal progression of the application as each configuration will typically process data and then produce some intermediate result that serves as the input to a succeeding configuration. This has a tremendous effect on the design process because all of the configurations must be carefully designed such that they step through the various phases of an application and communicate intermediate results with other configurations [64].

2.4 The Xilinx Virtex FPGA family

In this section a more detailed description of the structure of the Virtex FPGA family will be given based on the Virtex FPGA manual [145]. The manual contains the detailed description of the device, in this section only the features that are important regarding the thesis (e.g. partial reconfiguration) will be described.

2.4.1 Virtex architecture

Virtex devices feature a flexible, regular architecture that comprises an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs), all interconnected by a rich hierarchy of fast, versatile routing resources.

The basic building block of the Virtex CLB is the logic cell (LC). An LC includes

a 4-input function generator, carry logic, and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each Virtex CLB contains four LCs, organized in two similar slices, as shown in figure 2.8 [145].

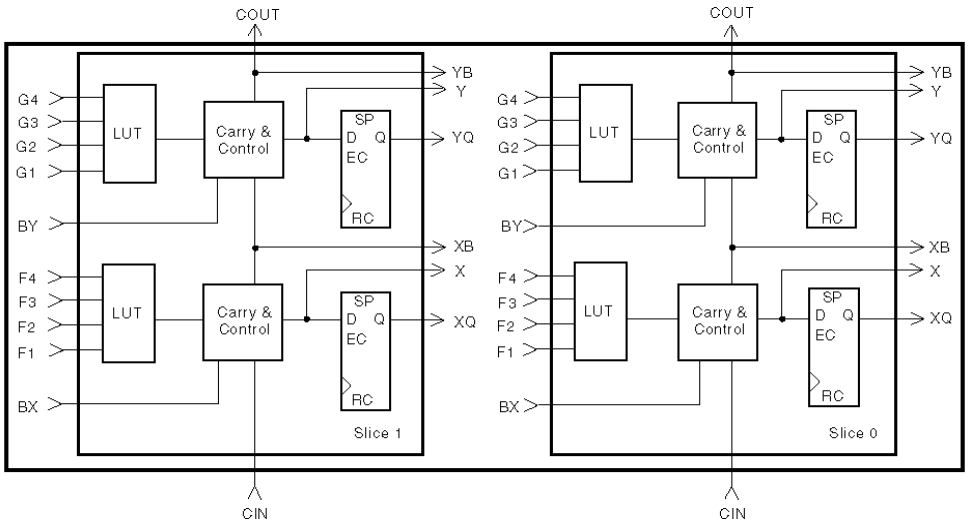


Figure 2.8: 2-slice Virtex CLB

Figure 2.9 shows a more detailed view of a CLB slice [145].

In addition to the four basic LCs, the Virtex CLB contains logic that combines function generators to provide functions of five or six inputs. Consequently, when estimating the number of system gates provided by a given device, each CLB counts as 4.5 LCs.

Virtex function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM. The Virtex LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing [145].

The storage elements in the Virtex slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing the function generators. In addition to Clock and Clock Enable signals, each Slice has synchronous or asynchronous set and reset signals (SR and BY). SR forces a storage

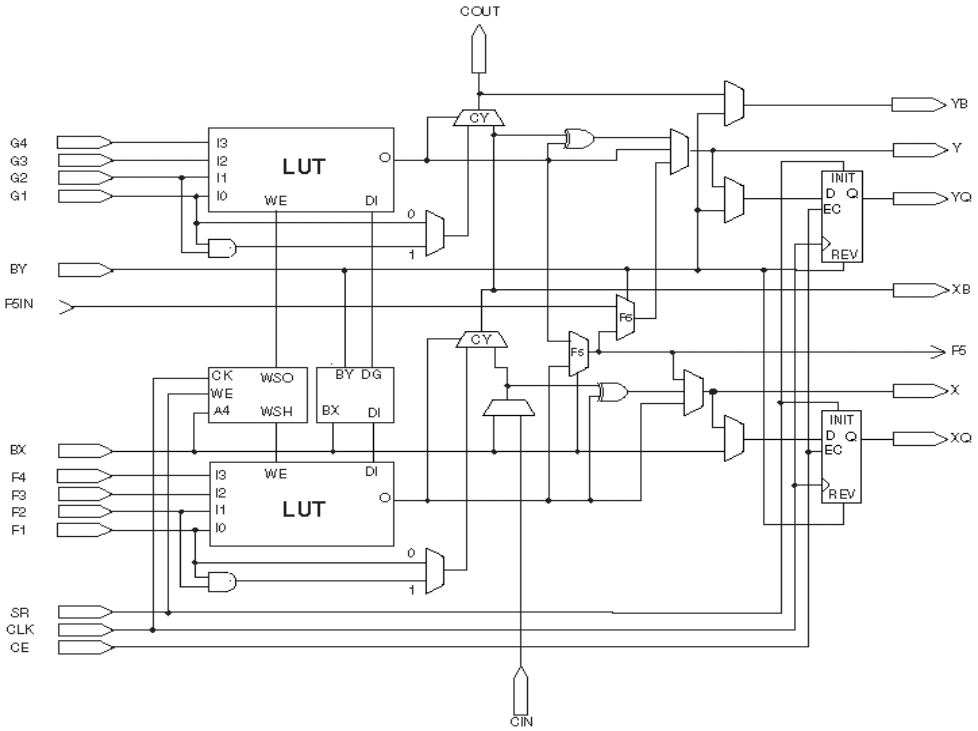


Figure 2.9: Detailed view of Virtex slice

element into the initialization state specified for it in the configuration. BY forces it into the opposite state. All of the control signals are independently invertible, and are shared by the two flip-flops within the slice [145].

2.4.2 Virtex reconfiguration

Each Virtex device contains configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs, clock resources, programmable routing, and configuration circuitry (Figure 2.10) [143]. These logic functions are configurable through the configuration bitstream. Configuration bitstreams contain a mix of commands and data. Configuration bitstreams can be read and written through one of the configuration interfaces on the device.

The Virtex configuration memory can be visualised as a rectangular array of bits. The bits are grouped into vertical frames that are one-bit wide and extend from the top of the array to the bottom. A frame is the atomic unit of configuration - it is the smallest portion of the configuration memory that can be written to or read from. Like this, partial reconfiguration/readback of the Virtex device can be made

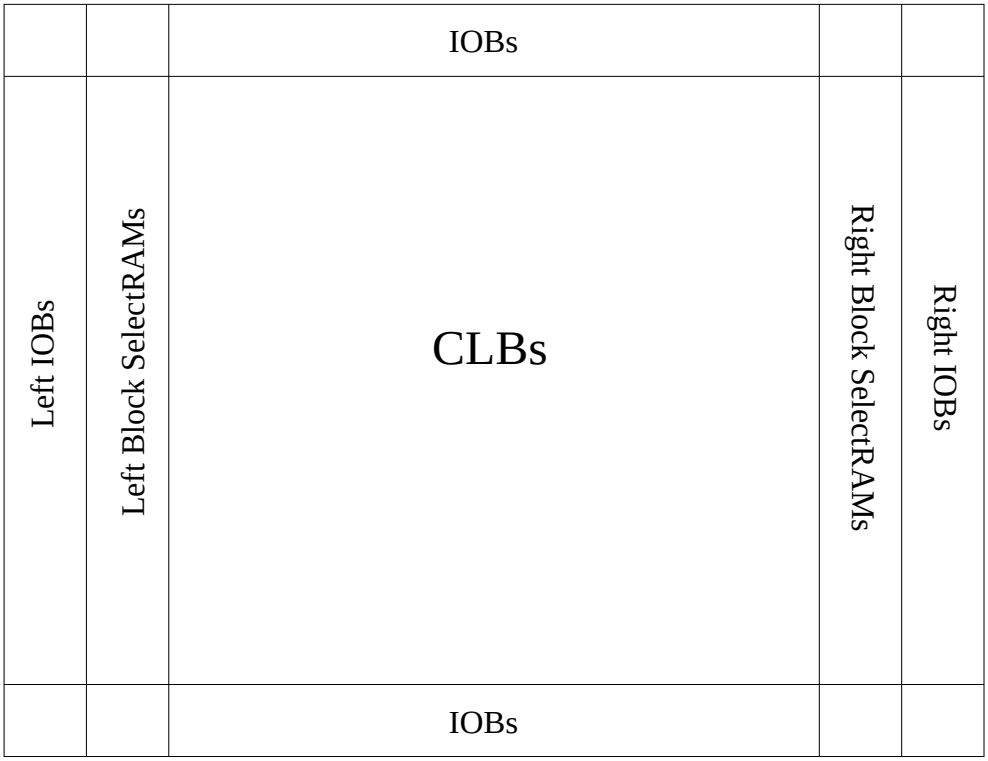


Figure 2.10: Virtex architecture overview

by frames. Partial reconfiguration can be performed with and without shutting down the device.

Frames are grouped together into larger units called columns. Table 2.3 shows the different kinds of configuration columns of Virtex devices [143].

Column type	**Number of frames**	**Number per device**
Center	8	1
CLB	48	Number of CLB columns
IOB	54	2
Block SelectRAM Interconnect	27	Number of Block SelectRAM columns
Block SelectRAM Content	64	Number of Block SelectRAM columns

Table 2.3: Configuration column types in Virtex devices

Each device contains one center column that includes configuration for the four global clock pins. Two IOB columns represent configuration for all of the IOBs on the left and right edges of the device. The majority of columns are CLB columns which each contain one column of CLBs and the two IOBs above and below those

CLBs. The remaining two column types involve the block RAM: one for content and the other for interconnect. For each RAM column, one of each type is present. The columns for a sample Virtex device are shown on figure 2.11 [145].

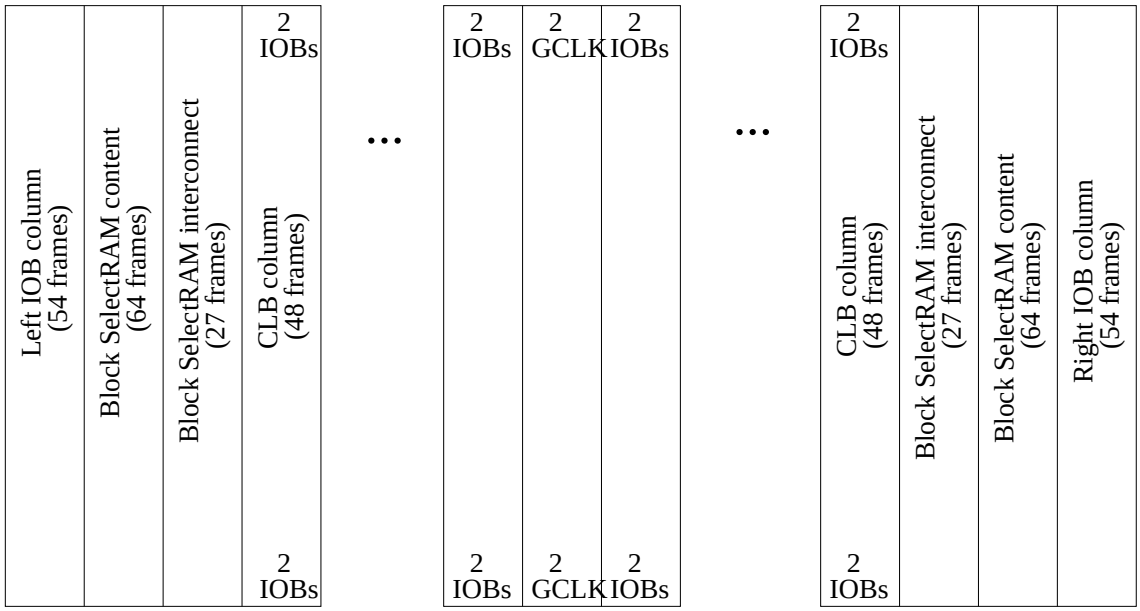


Figure 2.11: XCV50 device configuration columns

As it is mentioned above, the atomic unit of configuration is a frame, that means at least one frame must be reconfigured when partial reconfiguration is made. As a frame extends from the bottom to the top of the CLB array, each reconfiguration will touch all rows, but not certainly all columns.

There are a few design tools that support RTR and partial reconfiguration (e.g. JBits [59] [144] [58] or JHDL [17]). In this thesis work, the JBits API was used to develop a new fault injection methodology. This tool is described briefly in section 2.4.3.

2.4.3 The JBits API

The JBits API is a Java-based tool set, or application programming interface (API), that allows designers to write information directly to a Xilinx FPGA to carry out whatever customer logic operations were designed for it [59] [144] [58]. The JBits API permits the FPGA bitstream to be modified quickly, allowing for fast reconfiguration of the FPGA. With Virtex FPGAs, the JBits API can partially or fully reconfigure the internal logic of the hardware device [100].

The Virtex architecture allows this reconfiguration to be as extensive as necessary and still maintain timing information [59] [144]. The JBits API also makes it possible to integrate the operations of the FPGA with other system components such as an embedded processor, a graphics coprocessor, or any digital peripheral device.

JBits applications, or "applets", can use the Java API for Boundary Scan, unveiled by Xilinx, for platform independent device configurations deployed locally or remotely over the Internet [59] [144]. These applets can be control programs, consumer interface programs, or updates. Previously, Java applets were only used to send software updates via the Internet. The JBits API now makes its possible to create Java logic applets that can be used to send new hardware updates via the Internet.

To make an access directly to a board containing an FPGA device, JBits needs an interface towards this board. This interface is the Xilinx Hardware Interface (XHWIF) that is also developed in Java and contains all the information needed to communicate with the board from the host computer.

Some applications are known that have been developed with the JBits tool. Levi et al. developed a Java-based tool for evolving digital circuits on Xilinx XC4000EX and XL devices, using the JBits interface to control the bitstream configuration data and the XHWIF portable hardware interface to communicate with a variety of commercially available FPGA-based hardware [93]. James-Roxby et al. describe a tool constructed using JBits which extracts circuit information directly from configuration bitstreams and produces pre-routed and pre-placed cores suitable for use at run-time [69]. Patterson used JBits to implement the Data Encryption Standard (DES) in Virtex FPGA [111]. Scalera et al. prepared a plug-in for the Winamp MP3 player that uses a configurable computer [123] and used the JBits API. Ballagh et al. have realised behavioural hardware models using JBits that can be used in simulations [15]. Snider et al. developed a compiler that generates configuration bits directly from source code, and used JBits to generate the configuration bitfiles [129]. Sundararajan et al. presents a software-based technique for providing defect-tolerance for FPGAs using the JBits toolkit [130]. Detrey et al. describes the implementation of a core generator for arbitrary numeric functions in fixed-point format [48]. The implementation uses the JBits API to embed elaborate optimisation techniques in the description of the hardware. Lopez-Buedo et al. presents a new thermal monitoring strategy in [94] suitable for FPGA-based systems and all the hardware of the sensor

is written in Java using the JBits API. Carreira et al. used JBits to implement bit-serial LDI-ladder filters in FPGAs [30]. Robertson et al. proposes a method in [118] allowing the designer to apply standard hardware design and verification tools to the design of dynamically reconfigurable logic. In this approach, JBits is used for the generation of partial bitstreams.

2.5 Conclusion

Reconfigurable hardware possesses several advantages and some disadvantages as well with respect to ASICs. A very important advantage is that the same silicon can be reused for the execution of another application. On the other hand, timing characteristics of ASICs are better than the ones of reconfigurable hardware.

With the introduction of RTR the density of the circuitry is enhanced. Moreover, the reconfiguration process can be accelerated by applying partial reconfiguration of hardware resources.

This chapter gave an overview of reconfigurable devices, and focused on the fields that are the most important regarding the PhD thesis work, namely RTR and Virtex devices. The information presented in this chapter will be used in chapters 4 and 5.

Chapter 3

Fault injection techniques and limitations

In this chapter we give an overview and state of the art of the main fault injection techniques, targeting any type of circuits. The advantages and limitations of the different techniques are presented and compared. The techniques targeting a specific type of circuit (e.g., microprocessors) are not all considered here.

3.1 Introduction

Fault injection techniques have been proposed for a long time to evaluate the dependability of a given software, circuit or system implementation. Like this, fault injection can be hardware or software-based, or a mix of both methodologies. Most of the hardware fault injection approaches proposed up to now apply once the system or circuit is available. Such approaches include pin-level fault injection, memory corruption, heavy-ion injection, power supply disturbances or laser fault injection.

More recently, several authors proposed to apply fault injection early in the design process. The main approach consists in injecting the faults in high level models (most often, VHDL models) of the circuit or system. The injection of the faults can be done at different abstraction levels, from behavioural to gate-level descriptions.

In order to accelerate the fault injection process, it has been proposed to take advantage of hardware prototyping, using for example a FPGA-based hardware emulator. Another advantage of emulation is to allow the designer to study the actual behaviour of the circuit in the application environment, taking into account real-time interactions. When an emulator is used, the initial VHDL description must of course

be synthesizable. In some limited cases, the approaches developed for fault grading using emulators may be used to inject faults, but such approaches are classically limited to stuck-at fault injection. Specific approaches have therefore been proposed to inject other types of faults.

Section 3.2 deals with the hardware-based fault injection techniques. Software-based techniques are presented in section 3.3. Simulation-based techniques and hybrid (hardware- and software-based) methodologies are discussed in section 3.4. Finally, the different techniques are compared in section 3.5, a synthesis is presented and the reasons why a new methodology is proposed are described.

3.2 Hardware fault injection techniques

Hardware fault injection techniques are used to inject faults into actual hardware and to examine the effects on the system. Typically this is performed on VLSI circuits, because these circuits are complex enough to warrant characterization through fault injection rather than just a performance range. Faults are most often defined at transistor or gate level, because these are the best understood basic faults in such circuits [128]. Typical faults are stuck-at, bridging, or transient faults.

Hardware fault injections occur in actual samples of the circuit after fabrication. The circuit is subjected to some sort of interference to produce the fault, and the resulting behavior is examined. So far, this has been essentially done for external faults (permanent or transient) or for internal transient faults, as permanent stuck-at and bridging faults cannot be injected in the circuit without damaging it. The circuit is attached to a testing apparatus which operates it and examines the behavior after the fault is injected [128]. This consumes time to prepare the circuit and test it, but such tests generally proceed faster than simulation does and allow therefore a designer to analyse the effects of the faults when running a complex application.

It is possible to classify hardware fault injection techniques with respect to whether the injector has direct contact to the target system or not [63]. In case of injection with contact, the injector produces voltage or current changes externally to the target chip (e.g. pin-level fault-injection). When the injector has no direct physical contact to the target system, an external source produces some physical phenomenon, such as heavy-ion radiation or electromagnetic interference, causing spurious currents inside the target chip [63].

3.2.1 Pin-level fault injection

Pin-level fault injection is believed to have been the first technique used to inject faults. It employs specific hardware to change the electrical signals at selected target device pins. The chief versions of the technique apply either forcing probes or insertion sockets. The use of probes, which drive current on the target pin, produces stuck-at-0, stuck-at-1, and intermediate voltage level faults (a level that is neither a logical 0 nor a logical 1) [31]. The insertion approach requires target device(s) to be removed from the system board and replaced by a custom-made socket where digital logic intercepts each target pin. This technique enables injection of stuck-at faults as well as bit flips and logical bridging faults (AND/OR of adjacent pin signals). Both transient and permanent faults may be injected, and fault location and trigger can be set in a controllable and reproducible way. Trigger events may be set using target system hardware resources such as a clock signal or a bit-pattern in the data or address bus [31].

The key advantage of pin-level fault injection is its ability to inject close-to-real hardware faults. The technique is well suited to the evaluation of low-level dependability features such as hardware-based error detection mechanisms [31]. It can also target system components that are hard for other fault-injection techniques to access. A definite drawback, though, is the need for custom hardware. Moreover, with the rising levels of integration, the pin-level approach has lost some of its usefulness. Obviously, because of internal pipelines and caches, only a small part of what a complex processor does is visible at the pins [31].

The need for pin-level fault injection has arisen quite a long time ago. In [57] the testing of microprocessor system circuit packs is treated; the pin-level behaviour of the microprocessor is modeled in case of pin-level faults. A physical fault injection tool (MESSALINE) that uses pin-level fault injection is presented in [10, 8]. The validation of fault-tolerant mechanisms is addressed and the methodology is implemented on the general pin-level fault injection tool. In [97], physical fault injection at the pin level is used to evaluate the possibility to obtain a high-degree of fail-silent behaviour from a computer without hardware or software replication. [46] addresses the problem of processor faults in distributed memory parallel systems by injecting transient faults at the processor pins of one node of a commercial parallel computer. A similar problem is studied in [116] where error-detecting software techniques are evaluated in order to investigate if a non duplicated computer can be made fail-silent. To test

this, physical faults are injected to the pins of integrated circuits. Fault tolerance and error detection mechanisms in microprocessor and multi-processor systems are validated in [43, 44, 45] using physical pin-level fault injection. Another pin-level fault injection technique based on boundary scan design is presented in [34, 35] that is more efficient than the traditional hardwired pin-level fault injection.

3.2.2 Heavy-ion injection

Heavy-ion radiation is another way to inject faults by physically disturbing the hardware. In essence, a specific piece of hardware is bombed with heavy ions so that single-event upsets (SEU) occur in the digital circuits. As with pin-level methods, the greatest advantage of this technique is that close-to-actual hardware faults are injected [31]. In fact, components used in space are often hit by heavy ions and protons existing in radiation belts around the earth or elsewhere and are therefore specially designed, shielded, and tested against those particles.

Although a heavy-ion testbed is conceptually simple, its practical realization poses several problems. Special radioactive equipment is required, and coupling the bomber with the circuits may be difficult. Further, the great complexity and speed of today's processors present great obstacles to the design of the necessary monitoring hardware [31].

The injection of the faults itself is not an issue, but rather the difficulties of controlling and observing their effects inside the processor. In experiments using heavy-ion radiation equipment, the outputs of a target chip and a reference unit (or recorded reference behaviour) have to be compared pin by pin and cycle by cycle before it can be known whether or not the injected faults have produced errors inside the target chip [31]. A specific tester, such as the THESIC tester developed at TIMA [133] [2], can also be used to analyse the effects of the injections.

Heavy-ion fault injection was firstly introduced in [60], realised by a heavy-ion testbed used to evaluate the efficiency of several concurrent error detection schemes suitable for a watchdog processor. Soft errors were induced into a MC6809E microprocessor by heavy-ion radiation from a Californium-252 source, and recordings of error behaviour were used to characterize the errors and also to determine coverage and latency for the error detection schemes. In this environment, a miniature vacuum chamber is created by tightly coupling the irradiator to the packaging of the

target Motorola 6809 microprocessor of the board. The circuit under test is synchronized with an identical circuit not subject to irradiation, so as to compare their outputs. A host system controls the operation of the bomber as well as the collection of results [31].

Several experiences have been made using the environment described above. Evaluation of error detection mechanisms by two fault injection techniques, heavy-ion radiation and power supply disturbances is described in [82, 103]. Similarly, the validation of fault-handling mechanisms by heavy-ion radiation is described in [83] involving the injection of transient faults. The impact of physical injection of transient faults on processor execution is described and evaluated in [104] using the same environment and the same injection techniques, heavy-ion radiation and power supply disturbances.

A different environment, used to validate a fault-tolerant distributed real-time system (MARS), is described in [79, 80, 78, 81] and experimental results are presented. Three different fault injection techniques -pin-level fault injection, heavy-ion radiation and electromagnetic interference- are studied and compared in this environment, with emphasis on the specific impact of each fault injection technique and the effectiveness of error detecting mechanisms of MARS [49].

The evolution of technologies makes the circuits increasingly sensitive to protons and alpha particles even at sea level. Simple and cheap alpha sources can today be used to inject transient faults in circuits, in a way similar to the heavy ion injections. Here again, the most difficult part of the experiments is the analysis of the effects on the circuit behaviour.

3.2.3 Laser fault injection

This technique uses a laser to inject faults precisely onto internal nodes of a chip at specific times, allowing a higher level of control and a much better data set than the methods described in sections 3.2.1 and 3.2.2 [128]. The use of laser beams to induce Single Event Upsets (SEU) is not new [113, 25, 99, 24, 84, 124, 26, 101, 55, 74]. The laser fault injection technique has the advantage that it supports in situ testing of circuits and systems. It can also emulate transient error conditions that standard diagnostic test patterns and techniques may miss [68]. A very important advantage of laser fault injection is that faults are injected into the circuits in a non-intrusive, controlled manner. With heavy-ion radiation the upsets of specific functional elements

occur randomly. Inserting fault injection logic into circuits designs adds unwanted delays into high performance designs. In case of laser fault injection it is possible to upset specific gates or functional elements and to observe the response without degrading the performance of the design [68].

A pragmatic approach is presented in [134] injecting physical faults by means of a microcutting laser equipment on a set of good circuits in order to evaluate the efficiency of different test sequences. The experiments are performed on more than one hundred 68000 microprocessors. The injection of random spot defects at the circuit level by a laser-based equipment is described in [135] where the experiments were also carried out on 32-bit microprocessors. A technique is presented in [120, 106, 105] injecting soft, transient faults into VLSI circuits by laser fault injection in a precisely-controlled manner for the purpose of validating fault tolerant design and performance. The same technique is described in [119] offering the potential for performing automated testing of board-level and system-level fault tolerant designs including fault tolerant operating system and application software. In [28, 27], a pulsed laser was used to inject errors into an electronic system consisting of a number of different integrated circuits functioning as a digital version of an artificial neural network. The experiments were carried out in order to confirm that the system as a whole can operate autonomously in the radiation environment of space and to characterize the effects of the upsets on the output of the artificial neural network.

3.3 Software fault injection techniques

Software fault injection is used to inject faults into the operation of software and examine the effects. This is generally used on code that has communicative or cooperative functions so that there is enough interaction to make fault injection useful. All sorts of faults may be injected, from register and memory faults, to dropped or replicated network packets, to erroneous error conditions and flags [128]. These faults may be injected into simulations of complex systems where the interactions are understood but not the details of implementation, or they may be injected into operating systems to examine the effects.

Software simulation could be run on a high level description of a system, in which the protocols or interactions are known, but not the details of implementation. The injected faults tend to be mis-timings, missing messages, replays, or other communication faults. The simulation is then run to discover the effects of the faults. Because

of the abstract nature of these simulations, they may be run at a faster speed that the actual system might, but would not necessarily capture the timing aspects of the final system. This sort of testing would be performed to verify a protocol, or to examine the resistance of an interaction to faults. This would typically be done early in the design cycle so as to flesh out the higher level details before attempting the task of implementation. These simulations are non-intrusive, but they may not capture the exact behavior of the system [128].

Software fault injections can also be oriented towards implementation details, and can address program state as well as communication and interactions. Faults are mis-timings, missing messages, replays, corrupted memory or registers, faulty disk reads, and almost any other state the hardware provides access to. The system is then run with the fault to examine its behaviour. These simulations tend to take a longer time because they encapsulate all of the operation and detail of the system, but they will more accurately capture the timing aspects of the system. This testing is performed to verify the reaction of the system to introduced faults and catalog the faults successfully dealt with. This is done later in the design cycle to show performance for a final or near-final design [128]. These simulations can be non-intrusive, especially if timing is not a concern, but if timing is at all involved the time required for the injection mechanism to inject the faults can disrupt the activity of the system, and cause timing results that are not representative of the system without the fault injection mechanism deployed. This occurs because the injection mechanism runs on the same system as the software being tested [128].

A software fault injection tool is presented in [76, 75]. The FERRARI tool is an automatic real-time fault and error injector which can evaluate complex systems by emulating hardware faults in software. It is possible to emulate permanent faults and transient errors with this tool.

A software-implemented fault injection system is presented in [19] that is suited for microprocessor-based boards. The system runs on two different units connected by a serial port interface: the host computer and the target board. The tool injects transient single bit-flip faults in the memory image of the process (data and code) and in the user registers of the processor.

When injecting faults in a software-implemented environment, it is possible to exploit certain features available in some of the microprocessors and microcontrollers. Such a methodology is presented in [114] where the Background Debugging Mode of

the MC68332 microcontroller is used to perform fault injection experiments.

3.4 Hybrid and simulation-based fault injection techniques

Fault injection techniques can be implemented in a hybrid way, involving both software (SW) and hardware (HW). There are several possibilities to integrate HW and SW systems to build a fault injection environment. One possible way is to combine a microprocessor-based system with some reconfigurable HW e.g. FPGA-based environment controlled by software. It is also possible to introduce the faults early in the design process, most often by taking advantage of hardware description languages (HDL), such as VHDL. In this case, one possibility is to realise simulation-based fault injections, that means the behaviour of the fault-free and faulty application is analysed by simulating the HDL code. An important drawback related to the use of simulations is the huge amount of time required to run the experiments when many faults have to be injected in a complex circuit. To cope with the time limitations imposed by simulation, it has been proposed to take advantage of hardware prototyping, using for example an FPGA-based hardware emulator. Another advantage of emulation is to allow the designer to study the actual behaviour of the circuit in the application environment, taking into account real-time interactions. In this case, the initial VHDL description must of course be synthesizable to be able to download it onto the hardware emulator.

In this section these kinds of fault injection techniques will be discussed. Simulation-based fault injection is not really a HW and SW implemented technique. However, it is directly related to HW prototyping, thus the HDL simulation-based approach is also described in this section.

3.4.1 Hybrid techniques

Hybrid techniques are generally applied in a microprocessor-based system combined with some other HW resources (e.g. FPGAs). In these methods, no HDL description of the target system is used ; the FPGA only helps the fault injection process, generally to speed it up, but the faults are injected into the microprocessor-based system.

A hybrid HW-SW fault injection system is described in [77]. The main goal of this

approach is to overcome the limitations of both HW and SW based methods. The logic for the HW fault injection circuitry is implemented using FPGAs, and the SW is an extension of FERRARI, a SW based fault injection system mentioned in section 3.3. The SW based fault/error injection module emulates hardware faults/errors through software, by executing sequences of instructions to emulate the hardware fault. The HW based injection module applies logic signals at selected lines and buses in the target system. The target system is a SUN SPARC1 system and the faults are injected to its SBUS while the system is running different applications. In this approach the main advantage of the use of FPGAs, i.e. the capability to reconfigure part of the system, is already pointed out.

A similar methodology is proposed in [20] and [18]. In this approach, a board containing an FPGA is used to speed-up the fault injection process. The target system is also a microprocessor-based system ; the faults are injected into it and its behaviour is observed by means of the FPGA board. The host computer stores the relevant information for the whole fault injection process, and the FPGA board activates the fault injection process at the proper time.

Hybrid hardware/software fault injection technique is also applied in [115] to demonstrate the effectiveness of a method able to provide a microprocessor-based system with safety capabilities by modifying the source code of the executed application. The injection of bit-flips can be realised by this tool in the program under test, in the code or data areas.

At higher level, a hardware-software co-design tool allowing fault injection experiments is proposed in [86]. This approach is based on model simulations, but targets both the hardware and software parts of the system. The internal model of computation of the co-design environment is based on co-design finite state machines (CFSM). Two kinds of fault injections are realised in the experiments: behavioural and architectural fault injection. In case of behavioural fault injection, the faults can be injected by modifying the sequence of events the CFSMs detect and emit, the value associated to events and the values stored in the internal variables of the CFSMs. Like this, three kinds of behavioural faults are defined: faults on input events, output events or internal variables. In case of input/output events, three kinds of fault models can be identified: deleting the event when it has to be detected/emitted, changing the value associated to the event before detecting/emitting it, or detect an event that has not been emitted that means emitting an event at an erroneous time

in the case of output events. Fault injection on internal variables consists in changing the value stored in the internal variable. In the case of architectural fault injection, the target system is modeled as a single CPU with an external memory storing the program and the data of the software partition, and a set of hardware components connected to the CPU via a bus and to each other via dedicated connections. The transient single bit-flips are injected in the memory cells storing the code and the data of the software part and in the registers of the CPU and the hardware part.

3.4.2 Simulation-based techniques

In this section, we consider the injection of faults only in the hardware elements of the target system.

As mentioned above, it has been proposed to realise the injection of faults early in the design process, in the circuit or system model described using a hardware description language (HDL). The advantage of this methodology is its flexibility, as the design can easily be changed in order to inject different faults at different places. Also, in case of HDLs, the same description can be used on different platforms if the platform is suited for the specific HDL. However, there is an important disadvantage of the methodology which is the huge amount of simulation time.

The simulation-based fault injection has firstly been proposed in [107, 117]. The built-in error detecting mechanisms of a 32-bit RISC are analysed; data and control-flow errors are detected with a parity checking technique and a built-in watchdog. Fault injection is realised in a register-transfer level VHDL model by toggling the value of a randomly chosen internal state element bit. The duration of the faults was one clock cycle, and only single-bit transient faults were injected.

A different simulation-based fault injection tool (MEFISTO) is presented in [71, 70]. Injection is realised in VHDL models at various levels of abstraction, like this the identification and validation of an abstraction hierarchy of a fault/error model is also made. Two categories of fault injection techniques are proposed: the first one realises the injection by modifying the VHDL description, the second one uses the built-in commands of the simulator. Modification of the VHDL model can be made in two ways: by a saboteur or by a mutant. A saboteur is an additional VHDL component that alters the value or the timing characteristics of one or several signals when activated. Contrarily, a mutant does not need additional components, as it is the mutation of existing components. The specific component is replaced by its

mutant. It behaves as the original component it replaces when the fault injection is inactive, and it imitates the behaviour of the component in presence of faults when the injection is activated. In case of using the built-in commands of the simulator, there are two possibilities: altering the value of either signals or variables in the VHDL description. Like this, the VHDL model itself must not be modified, but the applicability of these techniques depends much on the functionalities offered by the command languages of the simulators.

A tool (MEFISTO-L) is proposed in [23, 9] which is derived from MEFISTO. The main objective of this tool was fault tolerance validation. The role of this tool is to manage and control the whole fault injection campaign: to place fault injection components (saboteurs) in the VHDL model, to launch the simulation and to extract the results.

A similar tool (MEFISTO-C) to this one is proposed in [131, 50] which is an improved version of MEFISTO. Contrarily to MEFISTO-L, this tool does not apply saboteurs or mutants but uses the other way of fault injection of MEFISTO: it injects faults via simulator commands in signals and variables defined in the VHDL model, so that the VHDL description must not be changed. In addition, this methodology is compared with a physical fault injection technique (scan chain implemented technique) in [50]. As a result, only minor differences in the distribution of detected errors were observed for the two different techniques.

Another simulation-based methodology is presented in [32, 33]. This approach focuses on bridging (BRI) defects in CMOS circuits, especially shorts between logic nodes and power lines. The faults are modeled by saboteurs, as mutants are not suitable to model BRI faults because the modified description of one gate driving the shorted nodes depends on the input signals of the other gate also driving the bridged nodes. Sets of faults are injected into the original VHDL description as saboteurs, using additional signals (control lines) to activate each single fault in sequence. Moreover, it allows both single and multiple fault injection by simple control lines programming. During the experiments, a commercial VHDL simulator is used.

A defect oriented fault simulation technique is also proposed in [122, 121]. A mixed-level fault simulator is here presented implementing bridging and line-open faults using the Verilog language. Fault injection is performed exclusively on structural parts of the circuit. Verilog models for the bridging and line-open defects are proposed for intra-gate and inter-gate faults, using a pre-computed test view of each

library cell.

A fault injection technique into VHDL behavioural level models is described in [47]. This methodology is developed for dependability parameter estimation and the fault injection is realised at the instruction set architecture level of modeling. The model is a behavioural description of a processor, and stuck-at 0, 1 or x faults are injected on different signals.

VHDL simulation based fault injection techniques at gate, register and chip level are presented in [16, 54, 56]. Three important techniques are described and compared: simulator command based, mutants and saboteurs, injecting both permanent and transient faults. In the simulator command based technique, four fault models are used when transient faults are injected: stuck-at 0 or 1, indetermination, bit-flip and delay. The permanent fault models are: stuck-at 0 or 1, indetermination, bit-flip and open-line (high-impedance). In case of saboteurs the fault models are the same, plus short, bridging and stuck-open for the permanent fault model. The injection campaigns have been performed into the VHDL-model of a 16-bit fault-tolerant microcomputer system running two different workloads.

A methodology to inject Single Event Upsets (SEUs) is proposed in [132]. Transient SEUs are injected into VHDL high-level descriptions in order to estimate the circuit reliability. The primary outputs of the circuit are verified for each transient fault that affects the functional circuit operation.

A VHDL simulation based experiment is described in [3]. The system which is modeled in VHDL is a real-time communication microprocessor architecture used in automated light-metro systems, based on an MC68302. Faults are injected in the internal registers of the processor, in the external memory and on the buses using the code alteration technique. The fault model is the single bit-flip fault; the induced errors match the characteristics of the errors produced by alpha particle hits. The injection is realised by additional fault injector elements that are inserted into the CPU, memory and bus modules. These added parts are activated by another additional element, a fault injection activator.

In [110], a technique is proposed to accelerate the simulation time of the fault injection campaigns in case of simulator-based techniques. Speeding-up the injection operations is reached by two basic methods: analysing the faults to be injected in order to identify the final fault effect as early as possible, and by exploiting some standard commands of commercial VHDL simulators. The fault model is the single

transient bit-flip affecting the memory locations containing data and code as well as user-accessible registers in a microcontroller. First, a fault-free execution is made and information about the behaviour of the system and the state of the simulator is stored in a trace file. In a second step, the information stored in the trace file is analysed in order to remove the faults the effects of which on the system can be determined a-priori. Simulation time is shortened as the system is periodically compared with the results of the fault-free simulation, and the simulation is stopped as early as the effects of the fault on the system become known.

A fault injection technique into VHDL descriptions by mutants is proposed in [91]. The generation of the mutants is optimised in such a way that faulty behaviours representative of significant faults can be forced during the fault injection experiments. In addition, these mutated descriptions are optimised for synthesis onto some emulation hardware, so that both simulation and emulation can be used starting with the same modified description. In the proposed methodology, the faults are injected in the control parts of synthesizable VHDL descriptions in two places: the inputs of the state register of a finite state machine and the inputs of the similar element(s) generated from a register-transfer level control flowchart. The fault model is the transient bit-flip model.

Multi-level fault injection experiments based on VHDL descriptions are presented in [92]. SEU-like fault injections are realised in VHDL descriptions of digital circuits considering several circuit description levels as well as several fault modeling levels. The results show that an analysis performed at a very early stage in the design process can actually give a helpful insight into the response of a circuit when a fault occurs.

3.4.3 Hardware prototyping

As mentioned in 3.4.2, simulation-based fault injection techniques have got an important disadvantage: the huge amount of time needed for simulation, even when optimization techniques (as presented in 3.4.2) are used. To cope with this problem, it has been proposed to take advantage of hardware prototyping, using for example an FPGA-based hardware emulator. There is another advantage of hardware prototyping: the designer can study the actual behaviour of the circuit in the application environment (or in an environment close to the application environment), taking into account real-time interactions. When an emulator is used, the initial VHDL description must of course be synthesizable as it must be mapped, placed and routed into

the emulator (generally an FPGA device).

The use of reconfigurable hardware in fault emulation was firstly based on approaches for fault grading using emulators. These approaches are generally limited to stuck-at faults. This methodology is firstly introduced in [138, 139]: it is pointed out that hardware prototyping accelerates fault simulation, and an FPGA-based environment is presented.

A method using an FPGA-based emulation system for fault grading is proposed in [37, 38]. The injection of both static and dynamic faults is proposed, without the recompilation of the design. The methodology is limited to stuck-at faults on the inputs of the logic blocks (CLBs). Faults can be injected by directly manipulating the corresponding bitstream of the affected CLBs (with the information extracted at compilation) and then downloading the modified bitstream into the device. Like this, no recompilation is needed. An approach was proposed and validated through experiments to realise dynamic fault injection. The basic idea is to duplicate the function the fault should be injected into. One instance of the function contains the original function, while the other one contains the faulty function, that means the same function but with a fault in it. At a moment, only one of the instances is activated. When normal emulation is made, the correct function is executed. When the fault is injected dynamically, the faulty instance of the function is activated with a fault activation controller. Like this, additional hardware is needed for each fault to inject and the recompilation of the design is needed to add these elements. Experimental results show that this means a maximal increase of about 30 percents in terms of number of CLBs.

A similar methodology is proposed in [62] attempting to inject faults without recompilation. The original circuit is converted into a fault injectable circuit that is reconfigured into the FPGA device. Faults are injected by shifting the content of a fault injection chain in the fault injectable circuit. The fault injection is also based on adding extra elements to the original circuit called fault injection elements, and the injection is also limited to stuck-at faults.

An approach similar to the ones presented in [37, 62, 38] is proposed in [126, 127]. The circuit is expanded with fault injectors and then it is mapped to the FPGA device. Fault injectors are used for modeling stuck-at-zero and stuck-at-one faults, and a fault activator is needed to control the fault injectors. The idea of injecting the stuck-at faults is similar as in the approaches mentioned above: primitive gates are

used for this purpose. This method does not require a specific logic emulator. The expansion of a circuit using fault injectors and fault activator resulted in an overhead of FPGA resources 3-4 times higher than in the circuit without additional logic. A method is proposed in [125] to attain a better usage of FPGA resources for fault emulation. The resource usage is high because circuit nodes are assigned randomly to fault injectors. A technique for an optimised assignment of circuit nodes to fault injectors through improved FPGA partitioning, mapping, placement and routing is proposed that reduces the 3-4 times overhead to 2-3 times, with respect to the circuit without additional logic.

Another hardware emulator based approach (which is the closest to the methodology proposed in the thesis work) is described in [29]. A Serial Fault Emulation (SFE) method in which each faulty circuit is emulated separately has been applied to gate-level circuits for stuck-at faults and has been implemented on a custom hardware emulator (Meta System).

Circuit transformations for supporting the injection of single transient bit flips in emulated circuits are proposed in [41, 40, 39], that do not require the reconfiguration of the FPGA for each fault analysis. There are three main modules in the proposed system, like in most of the typical fault injection systems. A fault list manager is in charge of analysing the system and generating the list of the faults to be injected. For this, the gate-level system description and the existing input stimuli is needed. The second module, a fault injection manager selects a fault, injects it into the system and observes the results. The result analyser checks the data produced by the fault injection manager. The fault list manager and the result analyser have been implemented as software processes. The fault injection manager is implemented as a hardware/software system with the software partition running on the host and the hardware partition located on the FPGA board along with the emulated circuit. The transformation of the circuit is realised by the fault injection manager. The software partition of the fault injection manager modifies the circuit for supporting the injection, sends the input stimuli to the FPGA board and triggers the injection and receives the system (faulty) behaviour from the board. The hardware partition of the fault injection manager is a fault injection interface interpreting and executing the commands issued by the manager. To transform the circuit, additional instrumenting elements must be added to the circuit: a register storing the information about which flip-flops should be effected by a bit flip, and a logic for injecting the fault. The

injection is controlled by an enable signal. The experiments were carried out on a standard Pentium-class PC and on a board with an XCV1000 FPGA. Experimental results show that the proposed circuit transformation introduces a limited hardware overhead that decreases with the size of the considered circuits.

The technique proposed in [41, 40, 39] is extended in [42] in order to better support fault injection into microprocessor-based systems. The hardware partition of the fault injection system (the interface on the FPGA) recognises commands issued by the fault injection manager, as described above. In the system of [42], specific commands suited for microprocessors can be recognised by the interface like executing one or more processor instructions, reading or changing the content of a specific storage element. The additional elements supporting fault injection are specific to a microprocessor architecture: a memory stub logic to inject faults into the memory, apply an input stimuli and observe the results, and a bus stub logic for the same purpose. The experiments were carried out on a board equipped with an XCV1000E FPGA and the modeled microprocessor was an Intel 8051. The results show that fault injection experiments can significantly be accelerated by this technique.

A software and FPGA-based fault injection technique is proposed in [21]. The fault injection platform has been developed in a real industrial environment, and the applied fault model is the single bit flip. The goal of the fault injection campaign is to grade possible faults if the fault(s) produce(s) a difference on an output port of the design during the test bench or in the memory elements at the end of the test bench. Some control functions are implemented in an FPGA, the rest of the system is software-implemented, simulation-based.

Fault injection into high-level VHDL models and FPGA-based hardware prototyping for early failure mode analysis of a complex digital circuit is proposed in [89]. The method applies the generation of mutants in VHDL RT-level descriptions to inject bit flips. The generation of the mutants has been automated according to previous approaches for VHDL modifications, and the experiments were carried out using hardware emulation.

3.5 Comparison and synthesis of fault injection techniques, proposition of a new technique

In this section the different fault injection techniques described above (sections 3.2, 3.3 and 3.4) are analysed. The study focuses on the advantages and disadvantages of the different methodologies, and a comparison of the techniques is given. Finally, the reason why the new fault injection methodology was proposed is described.

The subject of the thesis targets fault injection into hardware, so these methodologies will be analysed and synthesized in the following sections. The classical software fault injection techniques are not compared to the other techniques as they are quite different.

3.5.1 Analysis of hardware-based techniques

First of all, let us consider the advantages of the hardware-based fault injection techniques.

All hardware-based techniques have got the advantage that close-to-real faults are injected in a (close to) real environment. For example, pin-level fault injection (section 3.2.1) can target components that are hard for other fault-injection techniques to access. Also, this technique is relatively well suited to the evaluation of low-level dependability features.

Heavy-ion injection (section 3.2.2) has got the same advantage as pin-level fault injection: close-to-actual faults are injected, as digital circuits used in space applications are often hit by heavy ions. Also, any parts of the target circuit can be reached by this technique. Similar characteristics are obtained for alpha particle injections.

Laser fault-injection not only permits the injection of close-to-real faults in real environments, but it also supports in situ testing of circuits and systems. With this technique, the injection can be made very precisely and in an easily controlled way. A key advantage of this technique is that faults are injected in a non-intrusive and controlled manner into the target circuit.

The basic disadvantage of most hardware-based techniques is that they are quite inflexible. Pin-level fault injection needs custom hardware environment. Also, with the increasing level of complexity of VLSI circuits, the internal parts of the circuit become more complex, and these parts are not accessible for pin-level fault-injection. For example, the events on the input/output pins of a Pentium-class microprocessor

is only a small part of the operations of the processor.

Similarly to pin-level fault injection, the need for custom environment is a definite drawback of heavy-ion radiation, too. Special radioactive equipment is required for the experiences, and the adaptation of the bomber to the target circuit may be difficult. The monitoring of the events when the experiences are made is not easy in case of heavy-ion injection either. Another disadvantage of heavy-ion injection is that the injection of the faults is quite hard to control. The faults occur in the specific elements randomly, and the outputs of the target chip must be compared with the outputs of a reference unit cycle by cycle and pin by pin, with a limited level of observability.

Concerning laser fault injection, a possible disadvantage is the relatively high price of the instrumenting environment, comparing to other techniques.

Finally, a common disadvantage of these techniques is that they can be applied only after manufacturing of the first circuits, leading to very late detection of unacceptable behaviours and thus very long and costly rework.

3.5.2 Analysis of hybrid and simulation-based fault injection techniques

The advantage of simulation-based fault injection is its flexibility. It is easy to inject any kind of faults into any part of the design, one must change only the initial description (generally a HDL description) of the design, or use simulator commands. The description is then simulated with a commercial simulator which is executed on a workstation, and the results of the experiences can be easily captured. Moreover, it is also possible to inject different faults with a single description: external control signals can be used to manage the fault injection process. One important disadvantage of simulation-based techniques is the huge amount of simulation time: they need much more time than hardware-based methodologies to perform the experiments.

The goal of hardware- and software-based (hybrid) techniques is to integrate the advantages of both techniques (software-based techniques does not mean here the classical software fault injection techniques, but the simulation-based methodologies). One of the most important hybrid methodology is hardware prototyping. With this approach, the faults are injected in the same way as in the case of simulation, but the application is executed in reconfigurable hardware. Like this, the flexibility of simulation-based techniques is captured, but the execution time is lowered as the

application is not simulated but emulated in hardware.

However, there is a disadvantage of hardware prototyping: the time consumed by synthesis and place & route of the design. For different fault injection campaigns, different fault models must be adapted to the design, so the original description must be modified. Like this, the modified description must be re-synthesized and re-placed & routed before downloading it to the reconfigurable device. The time consumed by these operations can be important in case of large descriptions. A certain flexibility can however be attained, as similarly to the simulation-based methods, control signals can be applied here too. In this case the synthesized, placed & routed description of the design is downloaded to the reconfigurable hardware, and the emulation of the fault injection process is controlled through external signals. However, due to the hardware limitations of the emulator (number of pins or number of gates), several modified descriptions may be necessary, each of them allowing the injection of a subset of the faults. In that case, each of these descriptions has to be independently synthesized, placed and routed.

3.5.3 Summary of the comparison of different fault injection techniques

The advantages and disadvantages of the previous techniques are summarised in table 3.1.

	Advantage	**Disadvantage**
Pin-level fault injection	Injection in real environment, fast	Inflexibility: need for custom hardware, not everything is accessible
Heavy-ion radiation	Injection in real environment of real faults, all parts are accessible, fast	Need for custom hardware, hard to control, low precision
Laser fault injection	Real environment: in-situ experiments, high precision, easy to control	Expensive
Simulation	Flexible, easy realisation of injection, early analysis	Slow, simulation in software (not in real environment)
Hardware-prototyping	Early analysis, faster than simulation, more flexible than hardware-based techniques	Requires synthesizable models, slower than hardware-based techniques, less flexible than simulation

Table 3.1: Advantages and disadvantages of different fault injection techniques

3.5.4 Idea of a new methodology

As described in sections 3.5.2 and 3.5.3, the two basic points of simulation-based methodologies and hardware-prototyping are speed and flexibility. Simulation is flexible but slow, hardware prototyping is faster but less flexible.

These two points are the two pillars to start research into the direction of a new approach. The goal is to take more advantage of both techniques: keep as much as possible from the fastness of hardware prototyping and the flexibility of simulation. The idea is to apply hardware prototyping so the application is emulated in (reconfigurable) hardware, but inject the faults in a manner that the initial description must not be changed and like this only one synthesis and place & route must be made for the initial circuit description, without instrumentation for fault injection.

The detailed realisation of this idea is described in chapter 4.

Chapter 4

A new methodology for fault injection

A new methodology for fault injection is proposed in this chapter. The idea and the realisation principles are described. The implementation is discussed for two types of fault models in the case of the Virtex FPGA architecture.

4.1 Introduction

In this chapter, the proposed new methodology for fault injection is described and applied for two different fault models requiring very different implementations. In section 4.2, the new methodology is derived to overcome the limitations of the previous approaches. The application on Virtex FPGAs is then discussed in section 4.4.

4.2 Proposition of a new approach

As mentioned in section 3.5.4, the basic aim of the new fault injection technique is to cumulate the advantages of simulation and hardware prototyping: flexibility and fastness. Of course, such a hybrid technique cannot reach the same level of the advantages as each of the "initial" techniques, but the goal is to get as much gain as possible.

4.2.1 Limitations of simulation-based techniques

The most classical flow of simulation-based fault injection methodologies is shown on fig. 4.1.

The design is generally described in a HDL (e.g. VHDL, Verilog). This description must then be modified in order to support fault injection (if simulator commands are

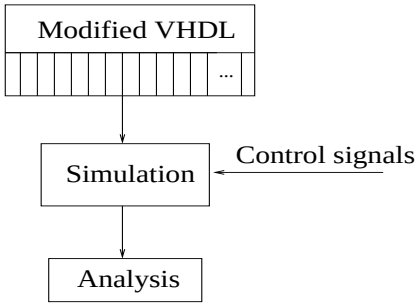


Figure 4.1: The flow of simulation-based fault injection

not used for that purpose), e.g. by adding mutants or saboteurs (section 3.4.2). Like this, one or several descriptions are generated for the different fault injection experiments. During the simulation of the modified description(s), fault injection can be controlled with external signals. These signals are activated in the testbench, in order to activate or stop a given fault injection. After simulation, an analysis can be made from the captured results.

As mentioned in section 3.5.2, a great advantage of simulation-based fault injection is its high flexibility. Faults can be injected everywhere in the circuit and a large set of fault models can be chosen by the designer.

The main disadvantage of simulation is the huge amount of time a simulation experiment needs. Depending on the complexity of the design, on the length of the testbench and on the number of faults to inject, a fault injection campaign can take several hours, days or even months to complete. This leads most often to the selection of a reduced set of faults (on either a deterministic or a random basis), compatible with the available time budget. However, the quality of the validation is of course reduced in the same proportion.

It must also be mentioned that a simulation testbench is in general quite far from a perfect modeling of the environment where the application will be executed. This limitation also reduces the quality of the analysis.

4.2.2 Limitations of classical hardware-prototyping

The classical flow of fault injection using hardware prototyping can be seen on fig. 4.2.

The beginning of the fault injection flow is basically the same as in the case of simulation: the initial description is changed in order to support fault injection or, in some cases, the modifications are made at gate level after synthesis of the initial

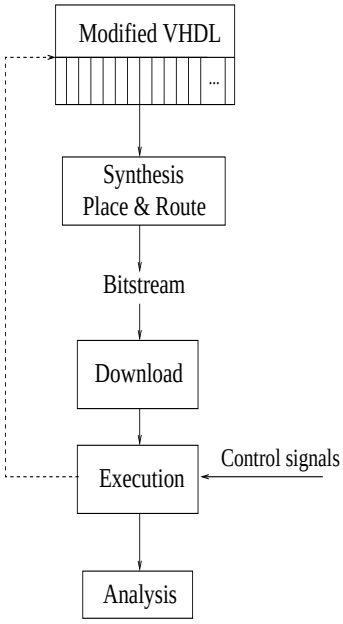


Figure 4.2: Fault injection with hardware prototyping

description.

Hardware prototyping however adds more constraints on the modifications than the simulation-based approach. Of course, the HDL description must be synthesizable. Furthermore, the number of gates and pins is limited by the available programmable device. Due to these hardware constraints, it is often not possible to make a single modified description that allows to inject all of the selected faults. The number of gates added to inject the faults is generally not the main limitation, but the number of inputs required to control the injection may often exceed the number of pins left on the device. Also, extra outputs can be required to observe internal nodes during the experiments, thus reducing further the number of control signals that can be added simultaneously. This may therefore lead to the generation of several modified descriptions, each of them allowing to inject a subset of the selected faults. These different descriptions must be separately synthesized, placed & routed and a bitstream is generated for each of them. Each bitstream can then be downloaded onto the reconfigurable device (generally an FPGA), for a given set of experiments. During the execution of these experiments, the fault injection is controlled through the external signals.

In this case, the execution of each experiment needs less time than with simulation-based techniques. Also, it is possible to connect the prototype in the real application environment (in-system emulation), that allows the designer to accurately analyse

the effect of faults at the system level. However, synthesis and place & route may have to be repeated many times, that can need a lot of time. Furthermore, the instrumentation in the circuit can have a noticeable impact on the critical path, reducing the maximum frequency at which the prototype can run, and therefore increasing the time needed for the experiments.

4.2.3 The proposed new methodology

The flow of the proposed new fault injection methodology is shown on fig. 4.3 [4] [5] [6] [90].

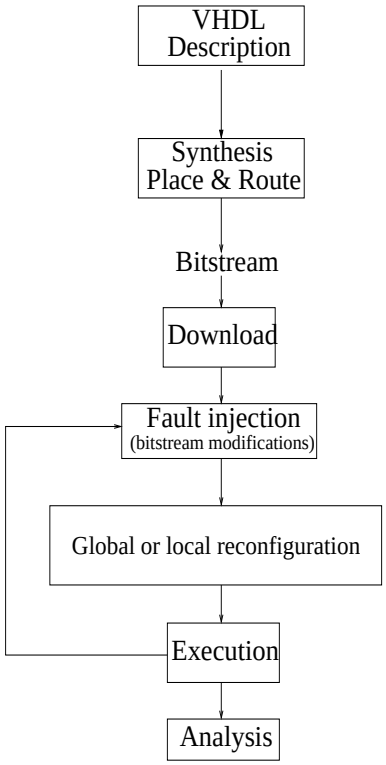


Figure 4.3: The flow of the proposed new fault injection methodology

The aim of the new fault injection technique is to try to avoid the problems of both simulation-based methods and classical hardware prototyping. First, it is therefore proposed to implement a prototype of the circuit without any instrumentation for fault injection. The initial HDL description is therefore directly used, without any modification, to generate a bitstream that can be downloaded onto the reconfigurable hardware (FPGA). Compared with the previous approaches, this leads to:

1. require only one synthesis and place & route process,

2. avoid any increase of the critical path,

3. avoid any additional input for injection control,

4. take advantage of the hardware speed,

5. keep the possibility of in-system emulation.

Then, the observation of internal nodes is performed using the readback capabilities of the programmable device. Like this, no additional I/O is required and the size of the FPGA can be kept in all cases at the minimum with respect to the size of the circuit under analysis. Finally, in the proposed methodology, the fault injection is realized at low level, by modifying directly part of the configuration bitstream instead of activating external control signals. This requires of course to define the bitstream modifications that are representative of the chosen fault model; this will be detailed in the following sections for two very different fault models.

In this new approach, a modified bitstream is therefore downloaded onto the FPGA at each injection time and the application execution is then resumed. Finally, the analysis of the obtained results can be made.

Since the FPGA must be reconfigured at each fault injection step, a lot of time would be lost if full (global) reconfigurations of the device are made. To avoid this, it is proposed to apply only a partial (local) reconfiguration, since only a small part of the configuration must be modified to inject a given fault [4] [5] [6] [90].

For a permanent fault, a single partial reconfiguration is therefore needed at the beginning of each experiment. For a transient fault, a first partial reconfiguration is done at the injection time and a second partial reconfiguration is done after a given number of cycles to remove the fault.

4.3 Description of the methodology

In this section, the application of the proposed fault injection methodology is described in detail. In section 4.3.1, the general description of the realisation is given. The injection of stuck-at faults is described in section 4.3.2. Finally, the description of the methodology to inject SEUs is given in section 4.3.3.

4.3.1 Overview, general description

In section 4.2.3 it is already mentioned that the bitstream obtained after synthesis, placement and routing is modified in order to inject the fault in the circuit description. For this purpose, the JBits API developed by Xilinx [144] is used (the JBits tool is described in section 2.4.3). The JBits API is not only used to change the bitstream; the reconfiguration of the FPGA and the readback of the bitstream to and from the FPGA are also made through this tool [4] [5] [6] [90].

The overview of the experimental environment is illustrated on figure 4.4.

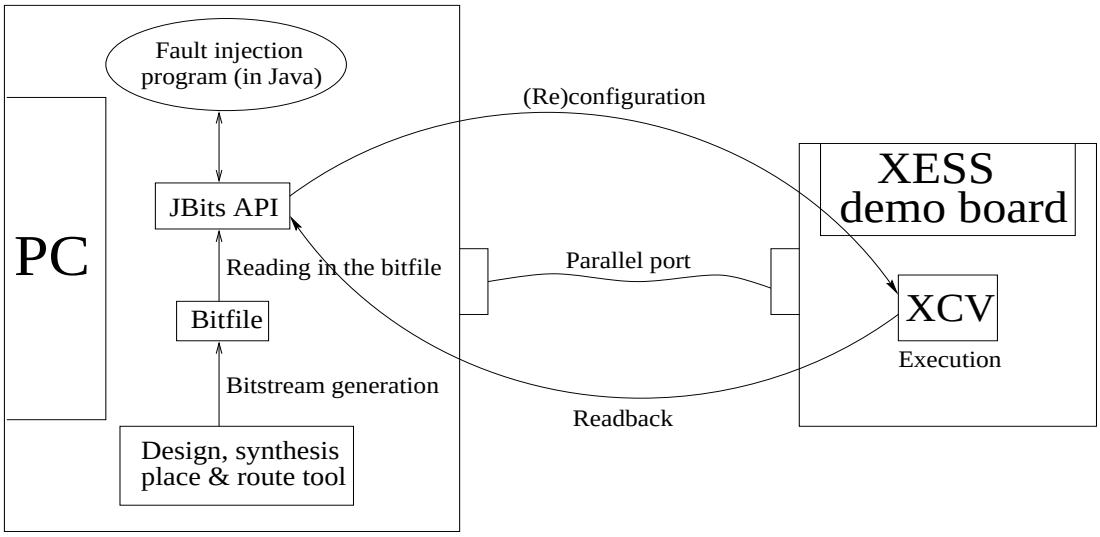


Figure 4.4: The fault injection environment

The first step is to synthesize the design (written generally in VHDL) and then generate a bitstream from it with some placing & routing tool. In our case, we generally used the Xilinx Integrated Software Environment (ISE) 4.1 and 5.1 tool [147] for synthesis and bitfile generation. The prototype was implemented on a Virtex development board from XESS [142].

The fault injection is managed by a program written in Java which manipulates the bitstream and communicates with the board through the JBits API. This program reads in the bitstream from a file, and then injects the fault(s) into it. The modified bitstream is downloaded by the program to the FPGA on the XESS board through the parallel port, and then the application is executed in the FPGA, also controlled by the fault injection program through the JBits API. At any time, the execution can be stopped, the design can be read back from the FPGA and its state can be analysed, or simply modified and reconfigured in order to inject a new fault.

The analysis can be made at the end of the fault injection campaign. To capture the data (the outputs of the design, the states of the flip-flops etc) there are different methods. For example, the states of the FFs can be captured by a readback. Output values can be stored in a register in the FPGA, and this register can be read at the end of the fault injection campaign.

Like this, the basic steps of the fault injection campaign of the proposed methodology are the following:

- reading in the initial bitstream from a file
- configuration of the device with the bitstream
- execution of the application
- fault injection (and/or removal)
- repeat the two previous steps until the end of the campaign
- analysis

Of course, at a fault-free execution of the application must be made in order to obtain the reference (nominal) values of the monitored signals.

Let us mention that the injection of a fault does not always simply mean the modification of the bitstream. For example, in the case of the injection of SEUs, the readback of the design is also needed (SEU injection is described in detail in section 4.3.3). Also, the modification of the configuration bitstream may be done in the bitstream file (that would then have to be downloaded into the FPGA), or, as in our case, directly in the circuit, in order to accelerate the reconfiguration.

4.3.2 Injection of stuck-at faults

In this section the injection of 3 kinds of faults is described: stuck-at-0, stuck-at-1 and inverting faults. Faults are called inverting faults when a signal/input has the inverse of the value the signal should be assigned to.

The technique of injection is the same for both faults [4] [5] [6] [90].

As described in section 2.4, the two most important elements (at least, from our point of view) of a CLB of a Xilinx Virtex FPGA [145] are the Look-Up Tables (LUT) and flip-flops (FF). When a VHDL (or other kind of) description is synthesized, placed

& routed, the combinatorial functions of the descriptions are mapped into LUTs. In case of the Xilinx Virtex family, the LUTs have got 4 inputs and 1 output, so any kind of logical function (Boolean expression) of 4 inputs can be realised. Like this, such a LUT can store 16 values as it has got 16 memory cells.

The targets of the injection must be identified in the mapped, placed & routed final design in the FPGA. These targets are chosen in the high-level HDL description as a net (signal, variable, ...) or a flip-flop. Their location must then be determined in the cell array of the FPGA. The Xilinx ISE tool generates a logic allocation file during place & route that allows to obtain these information; the FPGA editor of the Xilinx ISE tool can also be used to locate the targets of fault injection in the placed & routed design.

As far as combinatorial parts are concerned, the faults can only be injected on signals connecting CLBs, i.e. on LUT inputs or outputs. This is not a limitation in practice since injecting faults on other types of signals would be meaningless, due to the very different implementation in the final circuit and in the prototype. From a functional point of view, meaningful injection targets when injecting stuck-at faults classically correspond to primary inputs of the circuit, connected to the primary inputs of combinatorial blocks. In some cases, the injection can be useful also on a signal connecting two different blocks in the circuit description hierarchy. In all cases, these signals are mapped to LUT inputs if the synthesis maintains the description hierarchy. Of course, the injection of a fault on an intermediate signal that disappears during the synthesis optimizations is meaningless, since this signal will not exist in the actual circuit.

Once the targets localized in the FPGA array, the above-mentioned faults can be injected by modifying the contents of the LUT(s). Let us consider the example shown in Fig. 4.5, where the LUT contents are represented in a Karnaugh table. In this example a stuck-at-1 fault is injected on the F4 input of the CLB. It can easily be seen that in this case, the output value for F4F3F2F1=1000 must be copied to the output value for F4F3F2F1=0000, the value for 1001 to 0001, etc. ... The bitstream used for the experiment is modified accordingly.

Similarly, stuck-at-0 faults and inverting faults can also be injected on the inputs of CLB LUTs by changing the contents of the LUT. Injection of a stuck-at-0 fault on the F1 input is illustrated on figure 4.6. The LUT modifications in case of the injection of an inverting fault on the F2 input is shown on figure 4.7.

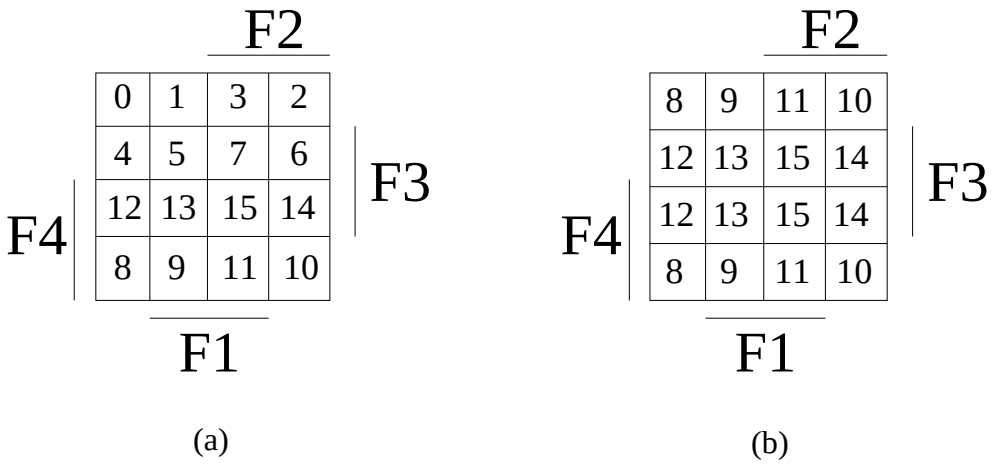


Figure 4.5: Injection of a stuck-at-1 fault on the F4 input of a CLB: (a) original LUT state and (b) modified LUT

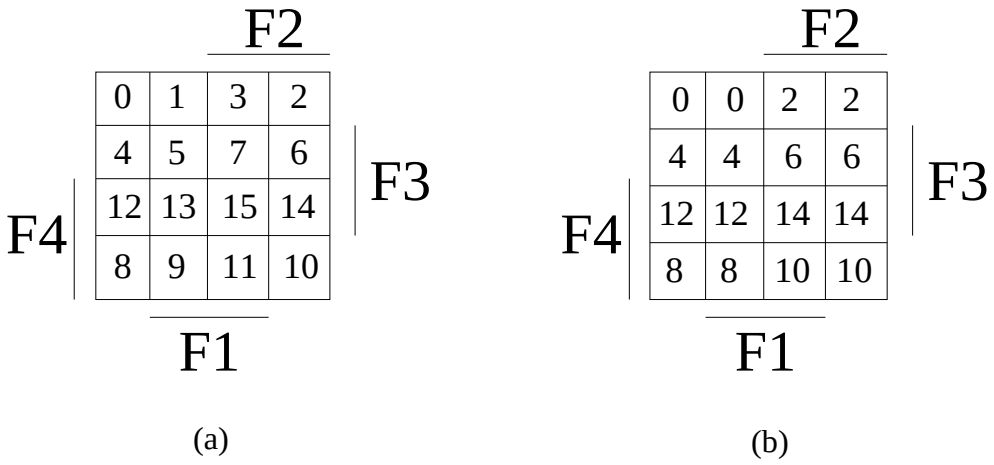


Figure 4.6: Injection of a stuck-at-0 fault on the F1 input of a CLB: (a) original LUT state and (b) modified LUT

As described in section 2.4, the Virtex FPGA family is organised by frames which extend from the bottom to the top of the CLB matrix, and a frame is the atomic unit of reconfiguration/readback. To inject a stuck-at-0 or stuck-at-1 fault, 8 bits in a LUT must be changed and these 8 bits must also be re-changed if the removal of the fault is needed. To change a bit in a LUT, 1 frame must be reconfigured. Like this, in case of stuck-at-0 or stuck-at-1 faults, a total of 16 frames must be reconfigured (if fault removal is needed, too).

The injection of inverting faults is slightly different, as in this case all LUT values (16 bits) must be changed, and re-changed also if fault removal must be done. Like this, 32 bits must be reconfigured in case of inverting faults if the fault must also be

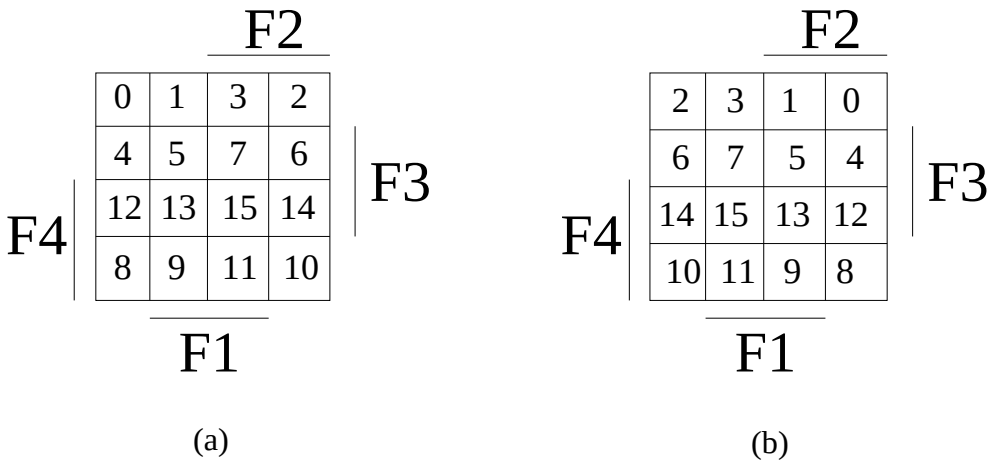


Figure 4.7: Injection of an inverting fault on the F2 input of a CLB: (a) original LUT state and (b) modified LUT

removed, which means the reconfiguration of 32 frames.

The structure of the program realising the injection of stuck-at faults is described on the figures in Appendix 6.

4.3.3 Injection of SEUs

An SEU, or Single Event Upset, corresponds to the erroneous modification of the contents of a memory element (memory cell, flip-flop, etc.) due to some external effect which can happen randomly (e.g. particle hits). In most cases, a single bit is modified (flipped) due to the event; most works in this area therefore aim at evaluating the impact of a single bit-flip on the behaviour of the system. Furthermore, due to the randomness of the events, the injection of a SEU means hereby that the state of the memory element changes asynchronously with respect to the clock applied to the element.

The injection of SEUs has recently become one of the most important field of fault injection. In chapter 3, some applications have already been mentioned in hardware-based techniques (section 3.2) and in hardware and software-based (mixed) methodologies (section 3.4) but many other research works can be found in this field.

This section shows a new methodology to inject SEUs, based on the same approach and environment as the stuck-at fault injection described in section 4.3.2: the Virtex family and the JBits API are used to carry out the experiments, and partial reconfiguration is also applied [6] [90]. However, the asynchronous modification of

the contents of a flip-flop cannot be realised by modifying the function in the combinatorial part; the SEU fault model therefore induces very different requirements than the fault models considered in the previous section.

The structure of the Virtex CLB is described in section 2.4. In this section, some characteristics will be analysed in more details from the point of view of SEU injection. In the Virtex family, it is not possible to "directly" change the state of a given flip-flop during the execution of the application on the prototype, so an indirect method must be applied. The only way to asynchronously change the state of a flip-flop is to apply a set or reset and like this, initialize it to "1" or "0". In the Virtex family, a net called GSR (Global set/reset) is used for this purpose which simultaneously initializes all flip-flops in the device. This GSR line can be pulsed using the JBits API.

For a given flip-flop in a Virtex CLB, the action of the GSR input can be set for two values: either it will apply a "set" to the flip-flop or a "reset" (section 2.4.1). The easiest way to describe it is to consider a switch which determines if a "set" or a "reset" is applied to the flip-flop when the GSR line is pulsed. The illustration is shown on fig. 4.8 [6] [90].

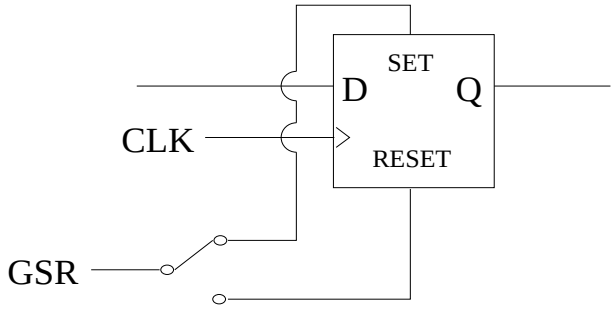


Figure 4.8: Simplified structure of a flip-flop in the CLB of a Virtex FPGA

This structure, as one can see, raises a problem. Pulsing the GSR line initializes all flip-flops (at least all flip-flops where this line is connected in). Like this, it can happen that not only the state of the target flip-flop will change but the state of all other flip-flops. To prevent this, some more operations must be made before pulsing the GSR line.

First of all, the states of all set-reset switches must be read at the beginning. The state of these switches will not change during execution, since the initialization value is fixed by design. At each fault injection time, the values of all flip-flops (or, at least all used flip-flops) must be read back. These states must be compared with the state of the corresponding switch. If the state of a flip-flop that is not the injection

target is "0" and the state of a switch is "reset", no modification needs to be done. Similarly, if the flip-flop is in state "1" and the switch is in "set", no changing should be done either. In the opposite cases (flip-flop at 1 and switch in reset or flip-flop at 0 and switch in set) the switch must be changed to its opposite polarity. Like this, if the GSR line is pulsed, the contents of the flip-flop will not be changed. Of course, the opposite configuration must be done for the injection target, as in this case the goal is to change the state of the flip-flop. After pulsing the GSR line (i.e., changing the state of the target flip-flop(s)), the switches that have been changed before the GSR pulse is applied must be re-changed, so that the initialization conditions of the flip-flops will not change when the application execution resumes.

The main steps of a SEU injection are therefore the following (considering that the states of the switches have been read and stored at the beginning of the campaign):

- read back the states of all used flip-flops
- compare the states of all used flip-flops with the corresponding switch states
- reconfigure the device to change the switches of the flip-flops where it is needed (in order to change the state of only the target flip-flop(s))
- reconfigure the device in order to pulse the GSR line
- reconfigure the device to re-change the switches that have been changed before GSR

Of course, the used flip-flops (including the target flip-flop) must be localised in the FPGA. It is possible to do it in the same way as described in the case of stuck-at fault injection (section 4.3.2), using the logic allocation file generated by the Xilinx ISE tool.

One readback and 3 reconfigurations are thus basically needed to inject an SEU into a flip-flop. In practice, the two first reconfigurations can be combined. However, the total injection process can result in quite a high amount of data to transfer between the host PC and the development board, that may need a relatively long time. This time should be reduced if not all the flip-flops and switches are read back and reconfigured, but only the ones actually used in the design under analysis. However, this reduction in the number of considered flip-flops does not systematically result in a proportional reduction in time. As it is described in section 2.4, a Virtex

FPGA is organised by frames that extend from the top to the bottom of the CLB matrix, and this is the atomic unit of reconfiguration/readback. Like this, the gain in time when only the used flip-flops are considered for reconfiguration/readback depends strongly on the placement and routing of the design. For example, if there is only one used flip-flop in a CLB column, its readback and reconfiguration for fault injection needs the same amount of data transfer as if all the corresponding flip-flops in all the corresponding CLB slices in the column had to be reconfigured/read back. For our application, this is a strong limitation of the Virtex architecture.

The structure of the program realising the injection of SEUs is described on the figures in Appendix 6.

4.4 Conclusion

The proposed new fault injection technique based on hardware prototyping was described in this chapter. The basic idea of the methodology is to carry out fault injection at low-level, directly in the bitstream, without any modification in the high-level or gate-level description of the circuit, and to take advantage of the partial reconfiguration capabilities of some FPGAs to reduce the injection time. The injection of faults can be achieved in the combinatorial parts as well as in the flip-flops; this has been detailed for several fault models on the example of the Xilinx Virtex architecture. The feasibility of the approach has been demonstrated on prototypes implemented on an XESS board, using the JBits API and a fault injection program developed in JAVA. The proposed approach implies to perform, for each fault injection, one or several partial reconfigurations (and also some readbacks in the case of SEUs), but considerably simplifies some other steps compared with previous techniques. The advantages and limitations are discussed in detail in the next chapter.

Chapter 5

Evaluations and results

In this chapter, calculations and experimental results are shown. The advantages and limitations of the proposed methodology are discussed and perspectives are drawn for further work.

5.1 Introduction

The methodology presented in chapter 4 is evaluated in this chapter by calculations and experimental results. In chapter 4, two different fault injection techniques are presented that need different realisations. The effectiveness of the methodology is analysed in this chapter for the two types of fault models.

The chapter is organised as follows. In section 5.2, the most important technological parameters of the Xilinx FPGAs are summarised, as a basis for the experiments and calculations. Section 5.3 deals with the calculations and the comparisons with previous approaches. Experimental measurements are presented in section 5.4. Finally, a possible optimised environment for fault injection experiments is proposed in section 5.5.

5.2 Technical background, limitations

When applying the proposed methodology, one must take into account the technological characteristics of the selected programmable device. In our case, the most important features are: the size of the CLB array of the FPGA (especially in case of global reconfiguration, but also in case of partial reconfiguration due to the organisation of the frames in a Virtex device), the structure of a CLB, the possible

execution and configuration frequencies, the number of bits in a full bitstream and in a frame, and the control mode of partial reconfiguration [90]. In this section these characteristics of the Xilinx Virtex FPGA family are summarised.

5.2.1 Technological details

The structure of the Virtex CLB is already described in section 2.4. The other most important characteristics (size and frequencies) are shown in table 5.1 [145]. Only the different devices of the Virtex FPGA family are examined as the experiments were carried out on a Virtex device. As mentioned in section 2.4.3, the JBits API supports only the Virtex family at the moment.

Device	**CLB array (rowxcolumn)**	**Maximal configuration frequency (MHz)**	**Maximal execution frequency**	**Number of configuration bits**	**Number of bits per frame**
XCV50	16x24	66	200	559200	384
XCV100	20x30	66	200	781216	448
XCV400	40x60	66	200	2546048	800
XCV1000	64x96	66	200	6127744	1248

Table 5.1: Characteristics of the different devices of the Virtex FPGA family

The frequencies are the same for each device as these technological characteristics are identical for all the devices in a given FPGA family. The configuration frequency can reach 66 MHz, but it is important to know that frequencies higher than 50 MHz need extra handshaking and the default configuration frequency is quite lower. On power-up, the configuration frequency is 2.5 MHz. This frequency is used until the ConfigRate bits have been loaded and define the frequency as the selected ConfigRate. Unless a different frequency is specified in the design, the default ConfigRate is 4 MHz. More details can be found in [145, 143, 146].

5.2.2 Reconfiguration and readback

Each Virtex device contains configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs, clock resources, programmable routing and configuration circuitry [143] organised in a matrix. Like this, the Virtex configuration memory can be visualised as a rectangular array of bits. The bits are grouped into vertical *frames* that are one-bit wide and extend from the top of the array to the bottom. A frame

is the atomic unit of configuration - it is the smallest portion of the configuration memory that can be written to or read from. This organisation therefore defines the constraints on both the partial reconfiguration and the readback steps. Further details about the configuration architecture of Virtex can be found in [143].

5.3 Effectiveness of the proposed methodology and comparison with other approaches

The calculations of the execution time of the proposed approach are presented in this section. The parameterised formulas to calculate the time needed for a fault injection campaign are given in section 5.3.1. Numeric calculations for the proposed methodology are presented in section 5.3.2. The sensitivity analysis with respect to the main parameters is detailed in section 5.3.3. The comparison of the proposed methodology with other methods (especially the simulation-based approach) is described in section 5.3.4.

5.3.1 Formulas to calculate the execution time of the proposed methodology

The execution time of a fault injection campaign is composed of different contributions in case of emulation-based fault injection. Basically, there are 5 different kinds of operations during the proposed emulation-based methodology:

- initialization: first (full) configuration of the device
- execution of the application on the device (clocking the FPGA)
- partial reconfiguration of the FPGA
- partial readback of the device
- computations of the program controlling the fault injection process

The input test patterns application and the reading of the outputs are not shown in the list above. In case of emulated fault injection, a simple manner to realise this is to implement some kind of pattern generator (e. g. memories) into the logic. In case of fault classification, the input test vectors can be read from such an element, and the output values can be analysed by some integrated mechanism. These elements

are added to the high level design, so they are configured into the device at the beginning, during the initialization step. The capture of the fault effects is made by the integrated monitor and transmitted to the host computer at the end of the whole process, that requires only a few bits to be sent. Like this, the related time can be neglected since this step must be made only once per experiment (or even once a campaign) in case of fault classification. The response monitoring would be more complex to treat and would have some impact on the duration of the campaign if a complete analysis of the fault propagations is required, leading to extensive data communications with the host computer to store all the intermediate responses of the circuit under analysis. However, this case is not treated hereby, as the aim of this chapter is to analyse the characteristics directly related to the use of run-time reconfiguration. Similarly, in case the test patterns have to be applied from the host computer, a noticeable amount of time may also be required to transfer the data between the computer and the prototyping board. However, this communication time would be the same in all emulation-based approaches using the same equipment. This parameter is therefore not taken into account in the following, and the application execution time is assumed to depend only on the prototype operating frequency.

The last operation of the list above is carried out on the microprocessor of the host computer, and in case of a Pentium-class CPU this time is proved to be completely negligible compared to the duration of the other operations. Like this, only the duration of the first 4 operations is taken into account in the following.

The initialization of the device means the full configuration of the FPGA with the bitstream corresponding to the design the faults have to be injected in. The duration of the initialization depends on the following parameters: configuration frequency ($f_{confreadb}$, that will be considered equal to the readback frequency), the total number of bits in the bitfile to be configured ($N_{bitstotal}$) and the number of bits loaded at a time ($N_{bitsparal}$) that means whether the device is configured serially or in parallel. Like this, the initialization time can be calculated by the following formula:

$$t_{init} = \frac{N_{bitstotal}}{f_{confreadb} N_{bitsparal}} \tag{5.3.1}$$

The execution of the application on the FPGA device is a quite simple operation with the assumptions discussed above: the device must be clocked after applying the application inputs. Like this, the time consumed by this operation depends on two factors: the frequency the device is clocked at (f_{exec}) and the total number of clock

cycles of the execution for the whole campaign (N_{cycles}). Like this, the formula to calculate the total time of execution is the following:

$$t_{exec} = \frac{N_{cycles}}{f_{exec}} \tag{5.3.2}$$

Partial reconfiguration and readback of the FPGA is needed in order to inject the faults into the circuit. In this chapter, only the injection of stuck-at faults and SEUs will be considered, but methodologies for the injection of other kinds of faults can be imagined on future research. The injection of different faults may need the reconfiguration and readback of different numbers of frames, depending on the nature of the fault and the realization of the injection. As mentioned above, one can assume that the frequency of reconfiguration and readback is the same ($f_{confreadb}$), like this there is no difference between readback and reconfiguration when calculating the time needed for it.

It is easy to deduce that the total duration of the injections strongly depends on the number of faults injected (N_{faults}). But the injection of a set of faults can necessitate the reconfiguration/readback of a different number of frames for the different faults. There are two reasons for it:

- the kinds of faults
- the placing and routing of the design

The first factor has already been discussed: the injection of different kinds of faults may necessitate the configuration/readback of different numbers of frames. For example, the injection of a SEU needs the reconfiguration/readback of more frames than the injection of a stuck-at fault.

The second factor is less easy to preview as it depends on the tool used for synthesis/place & route. The design can be placed and routed in different ways into the reconfigurable hardware. The placing of the resources and the occupation of the FPGA by the design can influence strongly the number of frames to be reconfigured/read back (section 4.3.3).

Like this, the total number of bits to reconfigure/readback during a fault injection campaign can be calculated from the following equation:

$$N_{bitsinject} = N_{bitsperframe} \sum_{k=1}^{N_{faults}} N_{framesfault_k} \tag{5.3.3}$$

where $N_{framesfaultk}$ is the number of frames needed to be reconfigured/read back to inject fault k and $N_{bitsperframe}$ is the number of bits in a frame (Table 5.1).

The duration of fault injection in a campaign can be deduced from equation 5.3.1 and 5.3.3:

$$t_{inject} = \frac{N_{bitsinject}}{f_{confreadb} N_{bitsparal}} = \frac{N_{bitsperframe} \sum_{k=1}^{N_{faults}} N_{framesfault_k}}{f_{confreadb} N_{bitsparal}} \tag{5.3.4}$$

From equations 5.3.1, 5.3.2 and 5.3.4 it is easy to deduce the total time needed for a whole fault injection campaign:

$$t_{total} = t_{init} + t_{exec} + t_{inject} \tag{5.3.5}$$

5.3.2 Numeric calculations

Calculations for stuck-at fault injections

Let us consider a simple example, i.e. the injection of stuck-at faults using an XCV50 device, with a configuration/readback and execution frequency of 1 MHz. We assume that 10,000 faults are injected ($N_{faults} = 10,000$), 1000 cycles are run during an injection experiment ($N_{cyclesperexperiment} = 1000$) and the configuration/readback is made 8 bits in parallel ($N_{bitsparal} = 8$). Let us consider the classical fault injection case, that means one experiment is required per injected fault and one experiment must be made for reference without fault. Like this a total of 10,001 experiments must be made in the campaign, and the total number of cycles can be calculated after the following equation:

$$N_{cycles} = (N_{faults} + 1) * N_{cyclesperexperiment} \tag{5.3.6}$$

When starting the fault injection campaign, the device must be initialized at the beginning, that means the full bitstream is configured into the FPGA. As shown on table 5.1, the number of bits in the bitstream of an XCV50 device is 559200. It is easy then to calculate the time needed to load the full bitstream to the device according to equation 5.3.1:

$$t_{init} = \frac{559200}{10^3 * 8} = 69.9ms \tag{5.3.7}$$

It is also possible to calculate the initialization time for each Virtex FPGA device, that will increase with the size of the FPGA. Table 5.2 shows these calculations, considering the same values for the operation (CCLK frequency, SelectMAP mode).

Device	**Initialisation time (ms)**
XCV50	69.9
XCV100	97.6
XCV400	318.2
XCV1000	765.9

Table 5.2: The time needed to configure different devices (with 8-bit parallel configuration at 1 MHz)

Let us now calculate the time needed for fault injection after equation 5.3.4 if partial reconfiguration is applied. As mentioned in section 4.3.2, twice the reconfiguration of 8 bits of a LUT of a CLB is needed for the injection of a stuck-at-0 or 1 fault. Partial reconfiguration is carried out by frames (section 5.2.2 and 2.4). One frame contains only one bit from a given LUT, for example it contains bit no. 0 from each F LUT of slice 0 for each CLB in the considered column. Like this, to reconfigure (or read back) 8 bits from a LUT, 8 frames must be reconfigured (read back). In this case, to inject and remove a stuck-at fault on the input of a LUT, 8 frames must be reconfigured twice ($N_{framesfaultk} = 16$).

Let us consider an XCV50 device. As shown on table 5.1, the frame of an XCV50 device contains 384 bits ($N_{bitsperframe} = 384$). Like this, the time needed to inject a single stuck-at fault on the input of a LUT can be calculated after equation 5.3.4 assuming that $N_{faults} = 1$, $f_{confreadb} = 1MHz$ and $N_{bitsparal} = 8$:

$$t_{injectonefault} = \frac{16 * 364}{10^3 * 8} = 7.28 * 10^{-1} ms \tag{5.3.8}$$

If 10,000 faults are injected, the total fault injection time is:

$$t_{inject} = 10000 * t_{injectonefault} = 7280 ms \tag{5.3.9}$$

As mentioned above, 10,001 experiments must be made and one experiment contains 1000 cycles. The total number of cycles can be calculated after equation 5.3.6:

$$N_{cycles} = (10000 + 1) * 1000 = 10001000 \tag{5.3.10}$$

The total execution time of the campaign can be calculated after equation 5.3.2 and 5.3.10:

$$t_{exec} = \frac{10001000}{10^3} = 10001ms \tag{5.3.11}$$

Considering equations 5.3.7, 5.3.9 and 5.3.11, the total time needed for the fault injection campaign in case of 10000 faults can be calculated after equation 5.3.5:

$$t_{total} = 69.9 + 7280 + 10001 = 17,350.9ms \tag{5.3.12}$$

Similarly to the calculations above, it is possible to calculate the time of the fault injection campaign in case of different circumstances (e.g. the number of faults differs, no partial reconfiguration is applied, etc.). The analysis of the impact of the different parameters on the effectiveness of the methodology is described in section 5.3.3.

Calculations for SEU injections

Similarly to the calculations of the previous section, it is possible to calculate the time needed for the injection of an SEU. As described in section 4.3.3, the values of some (in worst case, all) FFs and switches must be read and reconfigured, that means a higher number of data frames must be read and reconfigured at each injection step than in the case of stuck-at fault injection. As previously mentioned, all FFs must be read back once and all switches must be reconfigured twice in worst case. The readback of one type of FF of a whole column needs two frames, and a readback command must be sent before, that is 28 bytes in this case and leads to a total of 992 bits for one type of FF in one column. As there are 4 types of FFs in a column, and 24 columns in an XCV50 device, the number of bits read back in the whole device to get the states of all FFs is:

$$n_1 = 992 * 4 * 24 = 95232 \tag{5.3.13}$$

The reconfiguration of all the switches needs 11300 bytes. As it must be made twice, the total number of bits that must be reconfigured when modifying all the switches is:

$$n_2 = 11300 * 2 * 8 = 180800 \tag{5.3.14}$$

Let us consider the same environment and conditions as in section 5.3.2: 1 MHz configuration and readback frequency ($f_{confreadb} = 1MHz$), XCV50 device and the configuration/readback of one byte per CCLK cycle ($N_{bitsparal} = 8$). First of all, let us calculate the different time-properties of the methodology in *worst-case*, that means all the FFs must be read back and all the switches must be reconfigured twice at each injection step. In this case, the time needed to inject all SEUs can be calculated as follows:

$$t_{injectseu} = \frac{N_{faults}(n_1 + n_2)}{f_{confreadb} N_{bitsparal}} \tag{5.3.15}$$

Replacing the parameters of equation 5.3.15 by numeric values of the considered case, the injection time of one SEU ($N_{faults} = 1$) in worst-case conditions is given by the following equation:

$$t_{injectseu} = \frac{(95232 + 180800)}{10^3 * 8} = 34.504ms \tag{5.3.16}$$

Calculations for the injection of several numbers of faults in worst-case conditions are summarised in table 5.3 (also considering an XCV50 device). The execution time generally depends on the circuit complexity, as the length of a significant testbench increases with the number of functions in the circuit. Similarly, the number of faults to inject generally increases with the circuit complexity. Some examples are illustrated in the table.

As previously mentioned, all the above calculations in this section are *worst-case* calculations. Depending on the place & route tool and the occupation of the FPGA by the design, it is possible that only part of the FFs and switches must be reconfigured/read back (section 4.3.3). The worst-case described above is the case when 100% of the frames must be read, that means that the FFs used in the design are spread out all the FPGA array, and the switches that must be reconfigured at each injection step also require the maximum number of reconfiguration data. In addition, the reconfiguration/readback frequency was assumed to be quite low (1 MHz) with respect to the device capabilities.

	10,000 faults injected and 1,000 cycles per run	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
Initialisation (ms)	69.9	69.9	69.9
Execution time (min)	0.167	1.67	1,666.67
Fault injection time (min)	5.75	5.75	575.07
Total (min)	5.92	7.42	2,241.74

Table 5.3: Different times needed in worst-case for the fault injection process of SEUs when partial reconfiguration is applied

Let us now consider a better case: less than 75% of the device is occupied by the design, all used flip-flops being grouped in part of the columns so that only 75% of the FFs and switches must be reconfigured/read back at each injection. Also, let us assume a configuration/readback frequency of 10 MHz (the other parameters being the same as in the above examples).

Like this, the number of bits that must be read back becomes, after equation 5.3.13:

$$n_1 = 95232 * 0.75 = 71424 \tag{5.3.17}$$

Similarly, the number of bits to reconfigure in order to change the switches can be calculated from equation 5.3.14:

$$n_2 = 180800 * 0.75 = 135600 \tag{5.3.18}$$

Similarly to equation 5.3.16, the time needed to inject a single SEU can be calculated in these conditions after equation 5.3.15:

$$t_{injectseu} = \frac{(71424 + 135600)}{10^4 * 8} = 2.587ms \tag{5.3.19}$$

In this case, the time needed for the injection campaigns illustrated in table 5.3 is summarized in table 5.4.

Comparing tables 5.3 and 5.4, it can be seen that parameters modifications can easily result in a gain of one order of magnitude on the time needed for an injection campaign. The sensitivity of the various parameters on the duration of the campaign

	10,000 faults injected and 1,000 cycles per run	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
Initialisation (ms)	69.9	69.9	69.9
Execution time (min)	0.167	1.67	1,666.67
Fault injection time (min)	0.43	0.43	43.12
Total (min)	0.60	2.10	1,709.79

Table 5.4: Different times needed in better-case conditions for the fault injection process of SEUs when partial reconfiguration is applied

is therefore analysed in more details in the next section. Let us also notice from these first results that, in spite of the complex process for SEU injection due to the Virtex architecture, a large number of faults can be injected in a reasonable time. Even in the worst case considered in table 5.3, a full campaign based on a 100,000-cycle testbench and with 1 million faults injected only requires about one day and a half.

5.3.3 Sensitivity analysis

In this section, we analyse the impact of the main parameters on the time needed for a fault injection campaign on a Virtex-based platform. The studied parameters can be related to the basic characteristics of the prototyping environment ($f_{confreadb}$), to both these characteristics and the circuit characteristics (f_{exec}), to both these characteristics and the campaign definition ($N_{bitsinject}$), to the campaign definition alone (N_{faults}), or to the application run during each experiment ($N_{cyclesperexperiment}$). These analyses will be done in worst case from the placement and routing point of view, in case of SEU injection (that corresponds to the most complex process for run-time reconfiguration) and for a XCV50 device. The influence of N_{faults} and $N_{cyclesperexperiment}$ is strongly linked to the sensitivity analysis of the other parameters and will therefore be discussed at the same time, not in separate subsections.

The dependence of the campaign duration on the number of bits configured/read back in parallel ($N_{bitsparal}$) is not further analysed. This parameter has not got a wide range: either it is equal to 8 (when the SelectMAP mode is used [146]) or it is 1 (when the device is configured/read back serially). So let us consider $N_{bitsparal}$ to be constant, equal to 8 bits. Of course, the initialization time and the fault injection time would be multiplied by 8 in the other case, that would be inefficient for our

purpose.

Analysis of the impact of $f_{confreadb}$

As can be seen from equations 5.3.1 and 5.3.4, the initialization time and the injection time for the two types of fault models are inversely proportional to the reconfiguration frequency. The execution time is the only contribution independent from this parameter.

Assuming the cases studied in section 5.3.2, table 5.5 illustrates the evolution of the campaign duration with the reconfiguration frequency.

$f_{confreadb}$	**10,000 faults injected and 1,000 cycles per run**	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
500kHz	11.67	13.17	2,816.80
1 MHz	5.92	7.42	2,241.74
5 MHz	1.32	2.82	1,781.68
10MHz	0.74	2.24	1,724.18
20MHz	0.45	1.95	1,695.42
30MHz	0.36	1.86	1,685.84
40MHz	0.31	1.81	1,681.05
50MHz	0.28	1.78	1,678.17
60MHz	0.26	1.76	1,676.25

Table 5.5: Different campaign durations (in minutes) in worst case conditions for the SEU fault injections, as a function of $f_{confreadb}$

As can be seen from the table, the impact of $f_{confreadb}$ is not so significant (if at least at the order of magnitude of 1 to 10 MHz), except when a relatively small number of input patterns has to be applied with respect to the number of experiments (first case, with only 1,000 input patterns per experiment but 10,000 faults to inject). This parameter must therefore particularly be optimised when the number of faults to inject (i.e. the number of experiments) is high compared with the number of functional cycles in each experiment. This is the case when an exhaustive analysis is required, i.e. when all the faults that can occur during the application execution have to be injected. In the other cases, a medium frequency can be used without a noticeable impact on the campaign duration.

Let us also notice that it is almost useless to increase this frequency to its maximum. Due to the relationship between this frequency and the injection time (illustrated in figure 5.1), the actual reduction of the campaign duration becomes quite

small when the configuration frequency is pushed to its limit, although the design of the prototyping equipment may become much more difficult. On the opposite, reducing too much this frequency considerably increases the time spent for the injections.

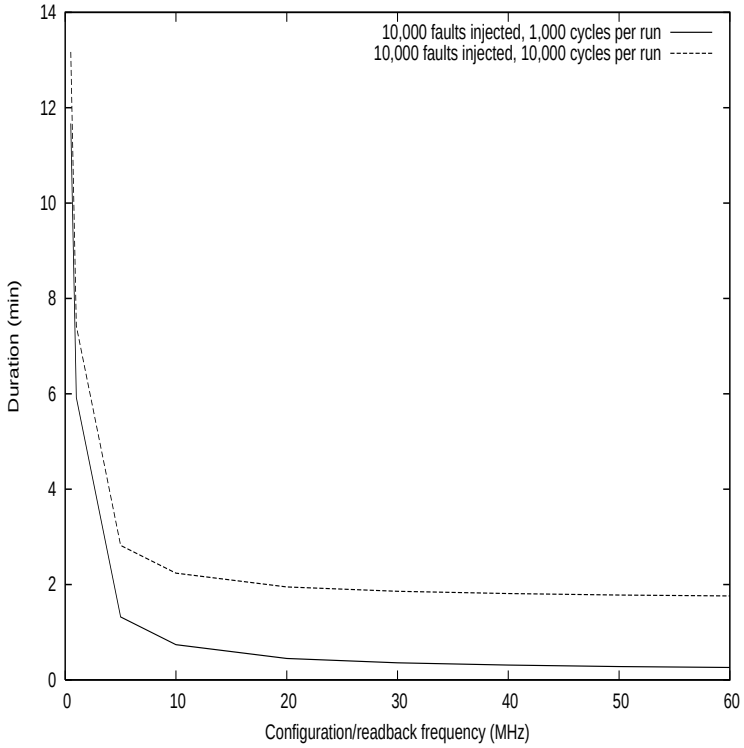


Figure 5.1: Graphical representation of the dependence of t_{inject} on $f_{confreadb}$, for the injection of an SEU: $N_{bitsinject}$=276,032 and $N_{bitsparal}$=8

Analysis of the impact of f_{exec}

On the opposite of the previous parameter, f_{exec} has only an impact on the execution time. This parameter depends on the basic speed of the programmable device and on the critical path in the circuit under analysis. Assuming the same cases as in the previous analysis, and a configuration frequency equal to 1 MHz, table 5.6 illustrates the evolution of the campaign duration with the execution frequency.

As can be seen from the table, the impact of f_{exec} can be very significant, if each experiment requires a large number of clock cycles, compared with the number of faults to inject. This is classically the case for a complex circuit, running a complex application and in which only randomly selected faults are injected.

The results in the table also illustrate the advantage of using hardware prototyping with a high system clock, and therefore the interest for high speed programmable

f_{exec}	**10,000 faults injected and 1,000 cycles per run**	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
500kHz	6.09	9.09	3,908.40
1 MHz	5.92	7.42	2,241.74
5 MHz	5.79	6.09	908.40
10MHz	5.77	5.92	741.73
50MHz	5.76	5.79	608.40

Table 5.6: Different campaign durations (in minutes) in worst case conditions for the SEU fault injections, as a function of f_{exec}

devices for such applications. Also, it would be worth spending some time in the initial synthesis and placement & routing process, so that the execution frequency can be increased. The use of run-time reconfiguration, avoiding any instrumentation of the prototype, leads to an increased maximal execution frequency and therefore helps to improve this parameter. Also, since only one synthesis and placement & routing is required, it is easier to spend time to optimise the result than in classical emulation-based approaches, in which this process may have to be repeated several times.

Analysis of the impact of $N_{bitsinject}$

Let us consider now that $f_{confreadb}$ and f_{exec} are fixed to 1 MHz. As described in section 5.3.2, the injection of a single SEU necessitates the reconfiguration/readback of up to 34 Kbytes, depending on the placement & routing of the design in a XCV50 Virtex device. Like this, the injection of 1,000 faults necessitates for example near 34 Mbytes to be transferred during the campaign for reconfiguration purposes.

Table 5.7 shows the evolution of the campaign duration if this amount of data could be reduced, by modifying for example the organisation of the frames and/or better controlling the placement process.

This table illustrates the very strong influence of the number of bits to be transferred. Due to the very small number of modifications actually required in the device to inject a fault, one could expect that only a few bits would have to be reconfigured. As explained in the previous chapter, the constraints imposed by the Virtex architecture lead in practice to a high transfer rate. Table 5.7 demonstrates that this is one of the most significant parameters, as one or two orders of magnitude of gain could be achieved on the campaign duration with a different architecture.

$N_{bitsinject}$	**campaign duration**
10 Kbytes	0.17
100 Kbytes	0.18
1 Mbytes	0.33
10 Mbytes	1.83
30 Mbytes	5.17

Table 5.7: Different campaign durations (in minutes) for the injection of 1000 SEU faults, in the case of 1,000 cycles per run, as a function of $N_{bitsinject}$

Let us recall that, in the case of stuck-at fault injection, the organisation of the frames in the Virtex devices requires to reconfigure 16 frames, i.e. 16 times 384 bits, when only 16 bits have actually to be modified to inject and remove the fault during each experiment. The results in table 5.7 also illustrate the impact of such a ratio between the useful bits and the transferred ones. With a better suited programmable architecture, a gain of at least an order of magnitude could be easily obtained on the campaign duration. The Virtex architecture has therefore the advantage to be a commercially available, partially reconfigurable architecture. But it is not well suited to the fault injection methodology proposed in this thesis.

5.3.4 Comparison with previous approaches

In this section, the numeric calculations of section 5.3.2 are compared with simulation-based experiments and "classical" techniques based on hardware prototyping. However, only the campaign execution time will be considered here. The different preparation times (including the synthesis, placement and routing steps) will not be taken into account. As previously mentioned, the time globally needed by such steps can be reduced in the case of the proposed approach compared with classical emulation-based approaches.

In case of "classical" fault injection techniques the description of the design is modified in order to be able to inject faults into it. This modified description is then either compiled (for simulation) or synthesized, placed & routed and downloaded into the configurable hardware (for emulation). To activate and control fault injection, external signals are used. It will be assumed in the following that this control does not require any additional cycle during the experiments, although it is not the case in all published approaches. On the contrary, in the methodology presented in this thesis, the duration of the campaign strongly depends on the number of injected faults as

each injection needs the partial reconfiguration of the device.

It is not easy to establish formulas to calculate the duration of a simulated fault injection campaign, so a parametric comparison of the different approaches is very difficult to realise. In order to be able to make some comparisons, the measures presented in [98] will be considered and compared to the results of calculations made for the hardware-based techniques. The experiments presented in [98] were realised with the ModelSim simulator on two types of workstations: SUN ULTRA10 and SUN ULTRA1. We will consider hereafter the most efficient case, i.e. the SUN ULTRA10. In that case, the average duration of a simulation cycle was found to be 0.24 ms. It should be noted that the measures in [98] have been obtained for a given circuit example, that is quite simple. The figures would have to be revisited for other circuit examples, and the duration of the simulations would increase in case of a more complex circuit. In consequence, the numeric values in this section just aim at comparing the order of magnitude of the campaign duration using the different approaches.

Let us consider the following conditions for the emulated fault injections: the device on which the experiments are carried out is an XCV50, the configuration, readback and execution frequencies are 30 MHz and the device is configured in parallel, 8-bits a time. In case of simulation, any number of faults can be injected at any cycle, almost without influence on the duration of the campaign. It is almost the same for "classical" emulation-based techniques where a signal is used to control fault injection: this operation needs almost no extra-time. The initialization of the simulation will be assumed to take about 500 ms, according to measures presented in [98]. The initialization of the hardware corresponds to the initial configuration of the FPGA and requires the same time as in the methodology proposed in this thesis. In the classical emulation-based approach, it has been mentioned that several instrumentations may have to be done. In that case, the initialization time has to be multiplied by the number of instrumentation versions. Since this is strongly dependent on the circuit under analysis and on the campaign definition, this will not be taken into account in the following comparisons and the initialization time will be considered only once.

In the comparison presented hereafter, it is considered that one SEU is injected at each fault injection step. We will consider a case in which a large number of faults has to be injected in a complex application, i.e. $N_{faults} = 1,000,000$ and

$N_{cyclesperexperiment} = 100,000$, as already considered in section 5.3.3.

The initialization time for the emulation-based fault injection campaigns can be calculated after equation 5.3.1 considering that $N_{bitstotal} = 559200$, $f_{confreadb} = 30MHz$ and $N_{bitsparal} = 8$:

$$t_{init} = \frac{559200}{3 * 10^4 * 8} = 2.33ms \tag{5.3.20}$$

The initialization time can be assumed to be the same for the proposed methodology and for the classical emulation-based approach, because the same hardware may be used to carry out the experiments. In case the logic added for instrumentation requires a larger FPGA to be used, the initialization time would be larger for the classical approach.

The fault injection time for the proposed methodology can be calculated according to equations 5.3.15 and 5.3.16. In the worst case from the placement and routing point of view, $n_1 = 95232$ and $n_2 = 180800$. Assuming $N_{faults} = 1,000,000$, $f_{confreadb} = 30MHz$ and $N_{bitsparal} = 8$:

$$t_{injectseu} = \frac{1,000,000 * (95232 + 180800)}{3 * 10^7 * 8} = 1,150.13s \tag{5.3.21}$$

The execution time for the proposed methodology can be calculated regarding equation 5.3.2 considering $N_{cyclesperexperiment} = 100,000$ and $f_{exec} = 30MHz$:

$$t_{exec} = \frac{(1,000,000 + 1) * 100,000}{3 * 10^7} = 3,333.34s \tag{5.3.22}$$

The execution time for the classical hardware-based approach is in best case the same. However, in practice, the maximum execution frequency is reduced by the instrumentation logic. To represent this increase in critical path, let us consider a 5% penalty on the execution time, that leads to a duration equal to 3,500 s.

Like this, the total duration of this fault injection campaign with the proposed methodology can be evaluated as:

$$t_{total} = t_{init} + t_{injectseu} + t_{exec} \simeq 75minutes \tag{5.3.23}$$

In the case of a "classical" hardware-based fault injection campaign, the duration would be smaller, since the increase in execution time is over-compensated by the

suppression of the injection time. The total time would thus be evaluated around 58 minutes. Let us however notice that this evaluation does not take into account the time for potential additional synthesis, placement and routing steps that may take hours. The proposed methodology may therefore be globally more efficient than the classical one. Furthermore, the worst case for placement and routing has been considered here. The gap between the two approaches would therefore be smaller if less frames are actually needed for the SEU injections.

In the case of a simulation-based campaign, and without taking into account specific optimisations mentioned in chapter 3, the time needed for the campaign would be, on the basis of the figures mentioned above, around 400,000 minutes, that means more than 9 months of simulations. Even considering the specific optimisations proposed in the literature, the duration of the campaign would remain much longer than in the case of the hardware-based approaches.

Let us now study how the campaign duration evolves with the number of faults to inject and the number of cycles per experiment. Table 5.8 shows the results in the three cases considered in section 5.3.3, with the same value as above for the other parameters.

approach	**10,000 faults injected and 1,000 cycles per run**	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
proposed methodology	0.20	0.25	75
classical emulation	0.006	0.06	58
simulation ULTRA10	40	400	400,000

Table 5.8: Comparison of campaign durations (in minutes) for simulation-based and emulation-based experiments in case of SEU injection

As illustrated by the results in this table, hardware-based fault injection is always more efficient than simulation. However, it is more significant in practice when the total number of cycles increases. Comparing the proposed approach and a classical emulation, the difference in terms of campaign duration increases of course (at the advantage of the classical approach) when the ratio between the number of faults to inject and the number of cycles per experiment increases. The proposed methodology is therefore better suited when each experiment requires a large number of cycles.

Let us now make a comparison of the same techniques, but in the case of stuck-at fault injection. As mentioned in chapter 4, the injection of a stuck-at-0 or 1 fault

requires the reconfiguration of 16 frames, i.e. 6144 bits, so the calculations will be made considering this value.

The values of t_{init} and t_{exec} will be the same as in the previous case, but the value of t_{inject} will noticeably differ. Using the previous numeric values for the different parameters, the injection time becomes:

$$t_{injectstuck-at} = \frac{1,000,000 * 6144}{3 * 10^7 * 8} = 25.6s \tag{5.3.24}$$

that leads to a total duration for the campaign evaluated as:

$$t_{total} = t_{init} + t_{injectstuck-at} + t_{exec} \simeq 56minutes \tag{5.3.25}$$

In the other approaches, the injection time is considered equal to zero and therefore the evaluation of the campaign duration does not change with the fault model. In consequence, for the stuck-at fault injection, the length of the campaign can be smaller with the proposed approach than with classical emulation, even considering the worst case for placement and routing.

Similarly to table 5.8, the total duration of the fault injection campaign can be calculated in case of different conditions. Some comparisons are shown in table 5.9, where the results for the other approaches are the same as previously.

approach	**10,000 faults injected and 1,000 cycles per run**	**10,000 faults injected and 10,000 cycles per run**	**1,000,000 faults injected and 100,000 cycles per run**
proposed methodology	0.01	0.06	56
classical emulation	0.006	0.06	58
simulation ULTRA10	40	400	400,000

Table 5.9: Comparison of campaign durations (in minutes) for simulation-based and emulation-based experiments in case of stuck-at injection

As can be seen in the table, the proposed approach compares much more favourably with the classical emulation-based approach in the case of stuck-at injection than in the case of SEU injection. This is coherent with the result of the sensitivity analysis, that showed the considerable impact of the parameter $N_{bitsinject}$.

5.4 Experimental measures

Experimental measures were carried out on the XSV50 board [142] that was used to demonstrate the feasibility of the proposed methodology. This development board is not designed to carry out experiments with high clock frequencies, and the downloading/reading back of the data is made through the parallel port of the PC that operates at a quite slow speed. Still more critical, the clock CCLK applied by the JBits interface to the board was found to be at 3.9 kHz only. It was shown in section 5.3.3 that a medium frequency was sufficient, but that a too small value considerably alters the efficiency of the proposed approach. With the board and interface available, the experiments could not show such results as described in section 5.3. Some experimental measures are shown in table 5.10. Column 1 shows the time needed for the initialization of the device. Column 2 shows the time needed for the injection of one SEU fault. Column 3 shows the execution time for a simple application. As one can see, the values measured are much higher than the values calculated in section 5.3, because of the low capabilities in speed of the XSV board.

Initialisation (ms)	**Fault injection (ms)**	**Execution (ms)**
19,678	3,566	55,961
19,688	3,575	55,873
19,718	3,498	55,912

Table 5.10: Measurements for the injection of a SEU on the XSV50 board (for three sets of experiments)

To achieve better performances, an application-specific environment must be built, trying to optimise the reconfiguration speed in order to be able to take advantage of the possibilities of our new fault injection approach. Such an environment is discussed in the next section.

5.5 A possible environment to realise optimised fault injection experiments

As mentioned in section 5.4, the XSV50 board is not an optimised environment to realise fault injection experiments that need a high reconfiguration and readback frequency. When using the JBits API, the communication between the board and

the PC is made through the parallel port. The XSV50 board needs a CPLD interface in order to support the JBits API: this interface is downloaded into the CPLD of the board, and this interface realises the communication between the parallel port and the FPGA device. All control, handshaking and clock signals are supplied via the parallel port and the CPLD.

The maximal bandwidth of a PC parallel port is 1Mbyte/s which is 8 Mbps. This bandwidth does not permit to supply high clock frequencies, that is why the configuration/readback frequency when using JBits is only about 4kHz. To overcome this limitation, a port with much higher bandwidth should be applied for the communication with the board.

The USB1.1 port has a slightly higher maximal bandwidth (12Mbps) which is still too small when a fault injection campaign should be made applying the proposed methodology. One possible solution would be the USB2.0 port which has got a maximal bandwidth of 480Mbps, or the Firewire port [65, 66, 67] with a maximal bandwidth of 400Mbps. With such a bandwidth, a configuration/readback clock of about 30MHz can easily be supported. The corresponding system architecture, optimised in bandwidth, is illustrated in fig. 5.2.

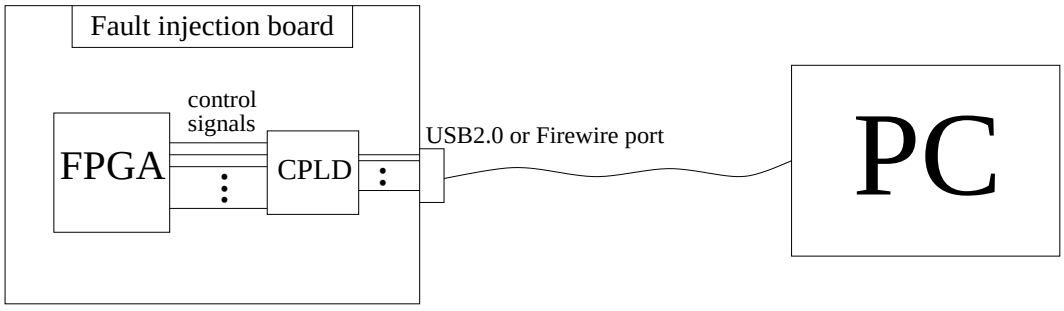


Figure 5.2: A possible optimised environment to carry out fault injection campaigns applying the proposed methodology

Of course, in this case a CPLD interface must be created which is able to handle the USB2.0 or the Firewire port. Also, it is possible that the JBits API on the PC must slightly be changed in order to communicate through the USB2.0 or Firewire port.

Apart from the bandwidth (directly related to the reconfiguration/readback frequency), there are other parameters that influence a lot the efficiency of the proposed methodology. For example, as described in section 5.3.2, the injection of a SEU necessitates the reconfiguration and readback of a huge amount of data: the state of

all (used) FFs must be read back and some (in worst case, all) of the switches must be reconfigured twice. This does not allow to take as much advantage of partial reconfiguration as expected.

A first possible way to improve this would be, in the case of a Virtex architecture, to define better suited placement and routing algorithms, aiming at minimising the number of frames that must be reconfigured for a given injection campaign. This would decrease $N_{bitsinject}$, that was shown to be a critical parameter.

Another way of reaching better performances would be to use better suited architectures than the Virtex one. Looking at other types of commercial FPGAs with partial reconfiguration capabilities (e.g. the AT6000 or AT40000 families from Atmel [11], [14]) may perhaps lead to some better results. However, commercial FPGA architectures target applications in which reconfigurations are quite seldom, on the opposite of the proposed methodology for fault injection. It can therefore be expected that only application-specific architectures would lead to the maximum speed of the fault injection campaigns. Developing such a new device would require a noticeable investment and was not the goal of this thesis, but may be a subject for further work.

Looking at existing architectures or at a completely new one, several characteristics can be optimised compared with the Virtex internal organisation.

Let us first consider the injection of a SEU. The method proposed to change the state of a FF is quite complicated, due to the constraints of the Virtex FPGA family architecture and of the JBits API. It can be imagined that the state of a FF could be changed with a simple configuration switch designed to change the state asynchronously, or at least activated individually without impact on the other flip-flops in the design. Like this, much less reconfiguration/readback steps would be needed, even no readback would have to be made. The number of bits to reconfigure/read back (n_1 and n_2 in eq. 5.3.13 and 5.3.14) would then decrease a lot. Consequently, $t_{injectseu}$ in equation 5.3.15 could become very small.

Similar optimizations could be proposed in the case of stuck-at fault injection. As mentioned in section 4.3.2 and 5.3.2, the injection and removal of a stuck-at fault necessitates twice the configuration of 8 frames. A different organisation of the frames may group all the bits in a LUT in the same frame. In that case, only two frames instead of 16 would have to be reconfigured to inject and remove a stuck-at fault. Also, reducing the size of a frame would lead to optimise the critical parameter $N_{bitsinject}$.

Chapter 6

Conclusion

A new approach for fault injection has been developed during the PhD thesis work. The main aspect of the approach was to apply hardware prototyping and RTR (especially, partial reconfiguration) in the fault injection process. Like this, not only the execution of the application is made in the reconfigurable hardware but also fault injection itself, at run-time. It avoids several runs of synthesis and place & route as the original (fault-free) design must be modified neither at high-level (e.g. in the VHDL description) nor at gate level to inject the faults. This can result in important time gains as place & route algorithms can take much time to complete.

The methodology was developed for two kinds of faults: stuck-at faults and SEUs. Stuck-at faults are injected by modifying the content of the LUTs in the FPGA CLBs. SEUs were induced in the flip-flops (FFs) of the FPGA by changing their state to the inverse with an asynchronous set or reset. The reconfiguration of the device can be made at run-time, by simply stopping the clock of the FPGA and reconfiguring part of it. This fault injection process has been automated. The execution of the application and the fault injection can be controlled from the same program which is executed on the host computer.

There are two main advantages of the proposed new methodology:

- time-gain
- no modifications of the initial design needed

As previously mentioned, important time-gains can be achieved by avoiding re-synthesis and re-placing & routing the design. On the other hand, each fault injection necessitates the (partial) reconfiguration of the device. In the case of stuck-at fault injection, global time gains can be achieved, even with so many reconfigurations, comparing to the "traditional" fault injection techniques. On the other hand, the injection

of SEUs necessitates the reconfiguration and readback of quite a huge amount of data, at least in the case of the Virtex architecture, and the reconfiguration time should therefore be lowered as much as possible. This can be made in different ways. One possibility is to optimise the fault injection environment by assuring a fast connection to the board containing the FPGA device. Another possibility would be to optimise the modification process of the FF contents. For this, an easier way to perform asynchronous modifications of the FFs should be provided by the FPGA manufacturer, or a specific programmable architecture should be designed.

For the above reasons, the injection of SEUs is not faster in all cases than the "traditional" techniques. It strongly depends on the size of the device and the design, the number of FFs used in the design, the way the manufacturer allows to modify the content of the FFs and the placing & routing of the design. Specific CAD tools with injection-oriented placement algorithms may also be a way to improve the time needed for fault injection. Let us also notice that the implementation of the circuit without any instrumentation leads to a prototype with a smaller critical path, that may allow to run the experiments with a higher system clock frequency.

One possible perspective of the methodology is to design an optimized environment, as mentioned above, to reduce reconfiguration time. Development boards, such as the one used during the PhD work, are generally not designed to realise long experiences, but rather to carry out tests of new designs or methodologies. In an optimized environment, the whole fault injection process (application of test vectors, data collection, analysis etc.) could be made on the same platform.

Another perspective of future research is to realise the injection of other kinds of faults using RTR (e.g. coupling faults).

Bibliography

[1] J. Abke, E. Böhl, and C. Henno. Emulation based real time testing of automotive applications. In *4th IEEE International On-Line Testing workshop*, pages 28–31, July 1998.

[2] M. Alderighi, F. Casini, S. D'Angelo, F. Faure, M. Mancini, S. Pastore, G. R. Sechi, and R. Velazco. Radiation test methodology for SRAM-based FPGAs by using THESIC+. In *IEEE On-Line Testing Symposium*, page 162, 2003.

[3] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, and M. S. Reorda. Fault behavior observation of a microprocessor system through a VHDL simulation-based fault injection experiment. In *Design Automation Conference*, pages 536–541, September 1996.

[4] L. Antoni, R. Leveugle, and B. Fehér. Using run-time reconfiguration for fault injection in hardware prototypes. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 405–413, October 2000.

[5] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection applications. In *IEEE Instrumentation and Measurement Technology Conference*, volume 3, pages 1773–1777, May 2001.

[6] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 245–253, October 2002.

[7] U. Apel. On-line software extension and modification. *Electrical Communication (Alcatel publication)*, 64(4), 1990.

[8] J. Arlat, M. Aguera, L. Arnat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injecton for dependanbility validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16:166–182, February 1990.

[9] J. Arlat, J. Boue, and Y. Crouzet. Validation-based development of dependable systems. *IEEE Micro*, 19:66–79, July-August 1999.

[10] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validaton of fault-tolerant computing systems. In *IEEE International Symposium on Fault-Tolerant Computing*, pages 348–355, June 1989.

[11] Atmel. *Coprocessor Field Programmable Gate Arrays, AT6000(LV) Series*, October 1999.

[12] Atmel. *AT94K Series Field Programmable System Level Integrated Circuit*, June 2002.

[13] Atmel. *AT94S Secure Series Programmable SLI*, June 2002.

[14] Atmel. *5K - 50K gates Coprocessor FPGA with FreeRAM AT40K Series*, 2003.

[15] J. Ballagh, P. Athanas, and E. Keller. Java debug hardware models using jbits. In *8th Reconfigurable Architectures Workshop*, April 2001.

[16] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil. A prototype of a vhdl-based fault injection tool. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 396–404, October 2000.

[17] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, April 1998.

[18] A. Benso, P. Civera, M. Rebaudengo, and M. S. Reorda. A low-cost programmable board for speeding-up fault injection in microprocessor-based systems. In *IEEE Reliability and Maintainability Symposium*, pages 171–177, January 1999.

[19] A. Benso, P. Prinetto, M. Rebaudengo, and M. Reorda. A fault injection environment for microprocessor-based boards. In *IEEE International Test Conference*, pages 768–773, October 1998.

[20] A. Benso, M. Rebaudengo, M. Reorda, and P. Civera. An integrated hw and sw fault injection environment for real-time systems. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 117–122, November 1998.

[21] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. New techniques for speeding-up fault-injection campaigns. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 847–852, March 2002.

[22] E. Böhl, W. Harter, and M. Trunzer. Real time effect testing of processor faults. In *5th IEEE International On-Line Testing workshop*, pages 39–43, July 1999.

[23] J. Boué, P. Pétillon, and Y. Crouzet. MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In *28th FTCS*, pages 168–173, June 1998.

[24] S. Buchner, K. Kang, W. J. Stapor, A. B. Campbell, A. R. Knudson, P. McDonald, and S. Rivet. Pulsed laser-induced seu in integrated circuits: a practical method for hardness assurance testing. *IEEE Transactions on Nuclear Science*, 37:1825–1831, December 1990.

[25] S. Buchner, A. Knudson, K. Kang, and A. B. Campbell. Charge collection from focussed picosecond laser pulses. *IEEE Transactions on Nuclear Science*, 35:1517–1522, December 1988.

[26] S. Buchner, J. B. Langworthy, W. J. Stapor, A. B. Campbell, and S. Rivet. Implications of the spatial dependence of the single-event-upset threshold in srams measured with a pulsed laser. *IEEE Transactions on Nuclear Science*, 41:2195–2202, December 1994.

[27] S. Buchner, M. Olmos, P. Cheynet, R. Velazco, D. McMorrow, J. Melinger, and R. E. J. D.Muller. Pulsed laser validation of recovery mechanisms of critical see's in an artificial neural network system. *IEEE Transactions on Nuclear Science*, 45:1501–1507, September 1997.

[28] S. Buchner, M. Olmos, P. Cheynet, R. Velazco, D. McMorrow, J. Melinger, R. Ecoffetand, and J. D.Muller. Pulsed laser validation of recovery mechanisms of critical see's in an artificial neural network system. In *IEEE European Conference on Radiation and Its Effects on Components and Systems*, pages 353–359, September 1997.

[29] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape. Serial fault emulation. In *IEEE Design Automation Conference*, pages 801–806, June 1996.

[30] A. Carreira, T. W. Fox, and L. E. Turner. A method of implementing bit-serial LDI ladder filters in FPGAs using jbits. In *IEEE International Conference on Field-Programmable Technology*, pages 433–436, 2002.

[31] J. V. Carreira, D. Costa, and J. G. Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum*, 36:50–55, August 1999.

[32] F. Celeiro, L. Dias, J. Ferreira, M. B. Santos, and J. P. Teixeira. Defect-oriented IC test and diagnosis using VHDL fault simulation. In *IEEE International Test Conference*, pages 620–628, October 1996.

[33] F. Celeiro, L. Dias, J. Ferreira, M. B. Santos, and J. P. Teixeira. VHDL fault simulation for defect–oriented test and diagnosis of digital ICs. In *IEEE–ACM European Design Automation Conference with EURO–VHDL and Exhibition*, pages 450–455, September 1996.

[34] S. Chau. Fault injection scan design for enhanced VLSI design verification. In *IEEE VLSI Test Symposium*, pages 109–111, April 1993.

[35] S. Chau. Fault injection boundary scan design for verification of fault tolerant systems. In *IEEE International Test Conference*, pages 677–682, October 1994.

[36] M. Chean and J. A. B. Fortes. A taxonomy of reconfiguration techniques for fault-tolerant processor arrays. *IEEE Computer*, 23:55–69, January 1990.

[37] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault emulation: a new approach to fault grading. In *IEEE-ACM International Conference on Computer-Aided Design*, pages 681–686, November 1995.

[38] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(10):1487–1495, October 1999.

[39] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting circuit emulation for fast hardness evaluation. *IEEE Transactions on Nuclear Science*, 48:2210–2216, July 2001.

[40] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga-based techniques for fault injection campaigns on vlsi circuits. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 250–258, October 2001.

[41] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Exploiting fpga for accelerating fault injection experiments. In *IEEE International On-Line Testing Workshop*, pages 9–13, July 2001.

[42] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante. Fpga-based fault injection for microprocessor systems. In *Asian Test Symposium*, pages 304–309, November 2001.

[43] C. Constantinescu. Validation of the fault/error handling mechanisms of the teraflops supercomputer. In *IEEE International Symposium on Fault Tolerant Computing*, pages 382–389, June 1998.

[44] C. Constantinescu. Using physical and simulated fault injection to evaluate error detection mechanisms. In *IEEE International Symposium on Dependable Computing*, pages 186–192, December 1999.

[45] C. Constantinescu. Teraflops supercomputer: architecture and validation of the fault tolerance mechanisms. *IEEE Transactions on Computers*, 49:886–894, September 2000.

[46] D. Costa, F. Moreira, H. Madeira, M. Rela, and J. G. Silva. Experimental evaluation of the impact of processor faults on parallel applications. In *IEEE Symposium on Reliable Distributed Systems*, pages 10–19, September 1995.

[47] T. A. Delong, B. W. Johnson, and J. A. Profeta. A fault injection technique for VHDL behavioral-level models. *IEEE Design and Test of Computers*, 13:24–33, Winter 1996.

[48] J. Detrey and F. de Dinechin. Multipartite tables in jbits for the evaluation of functions on FPGAs. In *IEEE International Parallel and Distributed Processing Symposium*, pages 154–160, April 2002.

[49] P. Folkesson. Assessment and comparison of physical fault injection techniques. Technical report, Chalmers University of Technology, Department of Computer Engineering, Fault Tolerant Computing for Embedded Applications Research Group, 1999.

[50] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *IEEE International Symposium on Fault Tolerant Computing*, pages 284–293, June 1998.

[51] E. J. G. and H. B. L. Density enhancement of a neural network using FPGAs and run-time reconfiguration. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180 –188, June 1994.

[52] E. J. G. and H. B. L. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable fpgas. In *IEEE World Congress on Computational Intelligence*, volume 4, pages 2097–2102, June 1994.

[53] E. J. G. and H. B. L. RRANN: the run-time reconfiguration artificial neural network. In *IEEE Custom Integrated Circuits Conference*, pages 77–80, May 1994.

[54] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil. A study of the effects of transient fault injection into the vhdl model of a fault-tolerant microcomputer system. In *IEEE International On-Line Testing Workshop*, pages 73–79, July 2000.

[55] C. Gossett, B. W. Hughlock, and A. H. Johnston. Laser simulation of single-particle effects. *IEEE Transactions on Nuclear Science*, 39:1647–1653, December 1992.

[56] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil. Comparison and application of different vhdl-based fault injection techniques. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 233–241, October 2001.

[57] J. Grason. Design aids and hardware testing of microprocessor system circuit packs. In *ACM IEEE Symposium on Design Automation and Microprocessors*, pages 95–99, 1977.

[58] S. A. Guccione. Reconfigurable computing at xilinx. In *IEEE Euromicro Symposium on Digital Systems and Design*, page 102, September 2001.

[59] S. A. Guccione, D. Levi, and P. Sundararajan. JBits: A java-based interface for reconfigurable computing. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, September 1999.

[60] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *IEEE International Symposium on Fault Tolerant Computing*, pages 340–347, June 1989.

[61] J. D. Hadley and B. L. Hutchings. Design methodologies for partially reconfigured systems. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–84, April 1995.

[62] J. H. Hong, S. A. Hwang, and C. W. Wu. An fpga-based hardware emulator for fast fault emulation. In *IEEE Midwest Symposium on Circuits and Systems*, volume 1, pages 345–348, August 1996.

[63] M.-C. Hsueh, T.-K. Tsai, and R.-K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30:75–82, April 1997.

[64] B. L. Hutchings and M. Wirthlin. Implementation approaches for reconfigurable logic applications. In *5th International Workshop on Field Programmable Logic and Applications*, pages 419–428, August 1995.

[65] IEEE. *IEEE 1394 standard for a high performance serial bus*, August 1996.

[66] IEEE. *IEEE 1394 standard for a high performance serial bus - Amendment 1*, 2000.

[67] IEEE. *IEEE 1394 standard for a high performance serial bus - Amendment 2*, 2002.

[68] J. J. R. Samson, W. Moreno, and F. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (lfi). In *IEEE International Symposium on Fault Tolerant Computing*, pages 162–167, June 1998.

[69] P. James-Roxby and S. Guccione. Automated extraction of run-time parameterisable cores from programmable device configurations. In B. Hutchings, editor, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 153–161, April 2000.

[70] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into (vhdl) models: A fault injection tool and some preliminary experimental results. In *IEEE International Workshop on Integrating Error Models with Fault Injection*, pages 13–14, April 1994.

[71] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *24th International Symposium on Fault-Tolerant Computing*, pages 66–75, June 1994.

[72] B. Jeong, S. Yoo, and K. Choi. Exploiting early partial reconfiguration of run-time reconfigurable fpgas in embedded systems design. In *Seventh ACM/SIGDA*

International Symposium on Field Programmable Gate Arrays (FPGA'99), February 1999.

[73] B. W. Johnson. *The Design and Analysis of Fault Tolerant Digital Systems.* Number 0201075709. Addison-Wesley Publishing, hardcover edition, January 1989.

[74] A. H. Johnston. Charge generation and collection in p-n junctions excited with pulsed infrared lasers. *IEEE Transactions on Nuclear Science*, 40:1694–1702, December 1993.

[75] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44:248–260, February 1995.

[76] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *IEEE Symposium on Fault-Tolerant Computing*, pages 336–344, July 1992.

[77] N. A. Kanawati, G. A. Kanawati, and J. A. Abraham. Dependability evaluation using hybrid fault/error injection. In *IEEE International Computer Performance and Dependability Symposium*, pages 224–233, April 1995.

[78] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. Integration and comparison of three physical fault injection techniques. Technical report, Chalmers University of Technology, Department of Computer Engineering, Fault Tolerant Computing for Embedded Applications Research Group, 1995.

[79] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Comparison and integration of three diverse physical fault injection techniques. Technical report, Chalmers University of Technology, Department of Computer Engineering, Fault Tolerant Computing for Embedded Applications Research Group, September 1994.

[80] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Evaluation of the mars fault tolerance mechanisms using three physical fault injection techniques. In *IEEE International Workshop on Integrating Error Models with Fault Injection*, pages 21–22, April 1994.

[81] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. Technical report, Chalmers University of Technology, Department of Computer Engineering, Fault Tolerant Computing for Embedded Applications Research Group, September 1995.

[82] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. Two fault injection techniques for test of fault handling mechanisms. In *IEEE International Test Conference*, page 140, October 1991.

[83] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14:8–23, February 1994.

[84] A. R. Knudson, A. B. Campbell, D. McMorrow, S. Buchner, K. Kang, T. Weatherford, V. Srinivas, J. G. A. Swartzlander, and Y. J. Chen. Pulsed laser-induced charge collection in gaas mesfets. *IEEE Transactions on Nuclear Science*, 37:1909–1915, December 1990.

[85] S. Y. Kung. VLSI array processors: designs and applications. In *IEEE International Symposium on Circuits and Systems*, volume 1, pages 313–320, June 1988.

[86] M. Lajolo, M. Rebaudengo, M. S. Reorda, M. Violante, and L. Lavagno. Evaluating system dependability in a co-design framework. In *IEEE Design, Automation and Test in Europe Conference and Exhibition*, pages 586–590, March 2000.

[87] R. Leveugle. Behavior modeling of faulty complex VLSIs: why and how? In *The Baltic Electronics Conference*, pages 191–194, October 1998.

[88] R. Leveugle. Towards modeling for dependability of complex integrated circuits. In *IEEE International On-Line Testing Workshop*, pages 194–198, July 1999.

[89] R. Leveugle. Fault injection in vhdl descriptions and emulation. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 414–419, October 2000.

[90] R. Leveugle, L. Antoni, and B. Feher. Dependability analysis: a new application for run-time reconfiguration. In *IEEE Reconfigurable Architectures Workshop (held with IEEE Parallel and Distributed Processing Symposium)*, pages 173–179, April 2003.

[91] R. Leveugle and K. Hadjiat. Optimized generation of vhdl mutants for injection of transition errors. In R. Reis, W. V. Noije, and J. Monteiro, editors, *IEEE Symposium on Integrated Circuits and Systems Design*, pages 243–248, September 2000.

[92] R. Leveugle and K. Hadjiat. Multi-level fault injection experiments based on vhdl descriptions: a case study. In *IEEE International On-Line Testing Workshop*, pages 107–111, 2002.

[93] D. Levi and S. Guccione. GeneticFPGA: evolving stable circuits on mainstream fpga devices. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *First NASA/DoD Workshop on Evolvable Hardware*, pages 12–17, July 1999.

[94] S. Lopez-Buedo, J. Garrido, and E. I. Boemo. Dynamically inserting, operating and eliminating thermal sensors of fpga-based systems. *IEEE Transactions on Components and Packaging Technologies*, 25:561–566, December 2002.

[95] P. Lysaght and J. Dunlop. Dynamic reconfiguration of field programmable gate arrays. In W. Moore and W. Luk, editors, *More FPGAs: International Workshop on Field Programmable Logic and Applications*, pages 82–94, Oxford, England, September 1993.

[96] P. Lysaght, J. Stockwood, J. Law, and D. Girma. Artificial neural network implementation on a fine-grained FPGA. In R. Hartenstein and M. Z. Servit,

editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. International Workshop on Field-Programmable Logic and Applications*, pages 421–431, Prague, Czech Republic, 1994. Springer-Verlag.

[97] H. Madeira and J. G. Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *IEEE International Symposium on Fault Tolerant Computing*, pages 350–359, June 1994.

[98] O. Malanda. Rapport de stage. Technical report, Techniques of Informatics and Microeletronics for computer Architecture (TIMA), September 2001.

[99] J. A. Mazer, K. Kang, and S. Buchner. Laser simulation of single-event upset in a p-well cmos counter. *IEEE Transactions on Nuclear Science*, 36:1330–1332, February 1989.

[100] S. McMillan and S. A. Guccione. Partial run-time reconfiguration using jrtr. In R. W. Hartenstein and H. Grunbacher, editors, *International Workshop on Field-Programmable Logic and Applications*, pages 352–360. Springer-Verlag, August 2000.

[101] J. S. Melinger, S. Buchner, D. McMorrow, W. J. Stapor, T. R. Weatherford, A. B. Campbell, and H. Eisen. Critical evaluation of the pulsed laser method for single event effects testing and fundamental studies. *IEEE Transactions on Nuclear Science*, 41:2574–2584, December 1994.

[102] T. Michel, R. Leveugle, G. Saucier, R. Doucet, and P. Chapier. Taking advantage of asics to improve dependability with very low overheads. In *European Design and Test Conference*, pages 14–18, February-March 1994.

[103] G. Miremadi, J. Harlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *IEEE International Symposium on Fault-Tolerant Computing*, pages 328–335, July 1992.

[104] G. Miremadi and J. Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44:441–454, September 1995.

[105] W. Moreno, F. Falquez, and N. Saini. Fault tolerant design validation through laser fault injection. In *IEEE International Caracas Conference on Circuits and Systems*, pages 132–137, March 1998.

[106] W. Moreno, F. Falquez, J. Samson, and T. Smith. First test results of system level fault tolerant design validation through laser fault injection. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 544–548, October 1997.

[107] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In *IEEE International Symposium on Fault Tolerant Computing*, pages 316–125, July 1992.

[108] I. Optimagic. Frequently asked questions (FAQ) about programmable logic. http://www.optimagic.com/faq.html, 2001.

[109] K. Parnell and N. Mehta. *Programmable Logic Design Quick Start Hand Book*. Xilinx, second edition, January 2002.

[110] B. Parrotta, M. Rebaudengo, M. S. Reorda, and M. Violante. New techniques for accelerating fault injection in vhdl descriptions. In *International On-Line Testing Workshop*, pages 61–66, July 2000.

[111] C. Patterson. High performance des encryption in virtex fpgas using jbits. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 113–121, April 2000.

[112] G. Pinter. Magasszintu hibamodellek validacioja. Master's thesis, Budapest University of Technology and Economics, 2002.

[113] W. D. Raburn, S. P. Buchner, K. Kang, R. Singh, and S. Sayers. Comparison of threshold transient upset levels induced by flash x-rays and pulsed lasers. *IEEE Transactions on Nuclear Science*, 35:1512–1516, December 1988.

[114] M. Rebaudengo and M. S. Reorda. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *IEEE VLSI Test Symposium*, pages 452–457, April 1999.

[115] M. Rebaudengo, M. S. Reorda, M. Violante, P. Cheynet, B. Nicolescu, and R. Velazco. System safety through automatic high-level code transformations: an experimental evaluation. In W. Nebel and A. Jerraya, editors, *Conference and Exhibition on Design, Automation and Test in Europe*, pages 297–301, March 2001.

[116] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *IEEE International Symposium on Fault Tolerant Computing*, pages 394–403, June 1996.

[117] M. Rimen and J. Ohlsson. A study of the error behavior of a 32-bit risc subjected to simulated transient fault injection. In *IEEE International Test Conference*, page 696, September 1992.

[118] I. Robertson, J. Irvine, P. Lysaght, and D. Robinson. Timing verification of dynamically reconfigurable logic for the xilinx virtex fpga series. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 127–135. ACM Press, 2002.

[119] J. Samson, W. Moreno, and F. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (lfi). In *International Symposium on Fault-Tolerant Computing*, pages 162–167, June 1998.

[120] J. R. Samson, W. Moreno, and F. Falquez. Validating fault tolerant designs using laser fault injection. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 175–183, October 1997.

[121] M. B. Santos, F. M. Goncalves, I. C. Teixeira, and J. P. Teixeira. Defect-oriented verilog fault simulation of soc macros using a stratified fault sampling technique. In *IEEE VLSI Test Symposium*, pages 326–332, April 1999.

[122] M. B. Santos and J. P. Teixeira. Defect-oriented mixed-level fault simulation of digital systems-on-a-chip using hdl. In *IEEE-ACM Design, Automation and Test in Europe Conference and Exhibition*, pages 549–553, March 1999.

[123] J. Scalera and M. Jones. A run-time reconfigurable plug-in for the winamp mp3 player. In B. Hutchings, editor, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320, April 2000.

[124] R. Schneiderwind, D. Krening, S. Buchner, K. Kang, and T. R. Weatherford. Laser confirmation of seu experiments in gaas mesfet combinational logic (for space application). *IEEE Transactions on Nuclear Science*, 39:1665–1670, December 1992.

[125] R. Sedaghat-Maman. Fault emulation with optimized assignment of circuit nodes to fault injectors. In *IEEE International Symposium on Circuits and Systems*, volume 6, pages 135–138, June 1998.

[126] R. Sedaghat-Maman and E. Barke. A new approach to fault emulation. In *IEEE International Workshop on Rapid System Prototyping*, pages 173–179, June 1997.

[127] R. Sedaghat-Maman and E. Barke. Real time fault injection using logic emulators. In *IEEE Asia and South Pacific Design Automation Conference*, pages 475–479, February 1998.

[128] R. Slater. Fault injection. Technical report, Carnegie Mellon University, 18-849b Dependable Embedded Systems, 1998.

[129] G. Snider, B. Shackleford, and R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *ACM International Symposium on Field Programmable Gate Arrays*, pages 115–124, 2001.

[130] P. Sundararajan and S. A. Guccione. Run-time defect tolerance using jbits. In *ACM International Symposium on Field Programmable Gate Arrays*, pages 193–198. ACM Press, 2001.

[131] S. Svensson and J. Karlsson. Dependability evaluation of the THOR microprocessor using simulation-based fault injection. Technical Report 295, Chalmers

University of Technology, Department of Computer Engineering, November 1997.

[132] F. Vargas, A. Amory, and R. Velazco. Estimating circuit fault-tolerance by means of transient-fault injection in vhdl. In *IEEE International On-Line Testing Workshop*, pages 67–72, July 2000.

[133] R. Velazco, P. Cheynet, A. Bofill, and R. Ecoffet. THESIC: a testbed suitable for the qualification of integrated circuits devoted to operate in harsh environment. In *IEEE Europen test Workshop*, Barcelona, Spain, May 1998.

[134] R. Velazco and B. Martinet. Physical fault injection: a suitable method for the evaluation of functional test efficiency. In *IEEE International Workshop on Defect and Fault Tolerance on VLS Systems*, pages 179–182, November 1991.

[135] R. Velazco, B. Martinet, and G. Auvert. Laser injection of spot defects on integrated circuits. In *IEEE Asian Test Symposium*, pages 158–163, November 1992.

[136] J. M. Voas and K. W. Miller. Examining fault-tolerance using unlikely inputs: turning the test distribution up-side down. In *IEEE Conference on Computer Assurance*, pages 3–11, June 1995.

[137] J. M. Voas and K. W. Miller. Using fault injection to assess software engineering standards. In *IEEE International Software Engineering Standards Symposium*, pages 139–145, August 1995.

[138] R. W. Wieler, Z. Zhang, and R. D. McLeod. Using an fpga based computer as a hardware emulator for built-in self-test structures. In *IEEE International Workshop on Rapid System Prototyping*, pages 16–21, June 1994.

[139] R. W. Wieler, Z. Zhang, and R. D. McLeod. Emulating static faults using a xilinx based emulator. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 110–115, April 1995.

[140] M. . J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99 –107, 1995.

[141] M. J. Wirthlin and B. L. Hutchings. Improving functional density using run-time circuit reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(2):247–256, June 1998.

[142] XESS Corporation, 2608 Sweetgum Drive, Apex NC 27502. *XSV Board V1.0 Manual*, September 2000.

[143] Xilinx. *Virtex Series Configuration Architecture User Guide*, September 2000.

[144] Xilinx, 2100 Logic Drive. San Jose, CA 95124-3450. *JBits 2.8*, September 2001.

[145] Xilinx. *Virtex 2.5V Field Programmable Gate Arrays*, April 2001.

[146] Xilinx. *Virtex FPGA Series Configuration and Readback*, November 2001.

[147] Xilinx. *Welcome to the Integrated Software Environment (ISE)*, 2002. http://toolbox.xilinx.com/docsan/xilinx5/help/iseguide/iseguide.htm.

[148] I. Xilinx. What is programmable logic? http://www.xilinx.com/company/about/programmable.html, 2001.

Publications

Using run-time reconfiguration for fault injection in hardware prototypes, *L. Antoni, R. Leveugle and B. Fehér*, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 25-27 October 2000, pages: 405-413

Using run-time reconfiguration for fault injection applications, *L. Antoni, R. Leveugle and B. Fehér*, IEEE Instrumentation and Measurement Technology Conference, Volume: 3, 21-23 May 2001, pages: 1773-1777

Using run-time reconfiguration for fault injection in hardware prototypes, *L. Antoni, R. Leveugle and B. Fehér*, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 6-8 November 2002, pages: 245-253

Dependability analysis: a new application for run-time reconfiguration, *R. Leveugle, L. Antoni and B. Fehér*, IEEE Reconfigurable Architecture Workshop (held with the IEEE International Parallel and Distributed Processing Symposium), 22-26 April 2003, pages: 173-179

Using run-time reconfiguration for fault injection applications, *L. Antoni, R. Leveugle and B. Fehér*, Accepted in IEEE Transactions on Instrumentation and Measurement Technology Conference, October 2003

Appendix

This appendix gives a brief overview of the program realising the injection of stuck-at faults and SEUs. The structure of the program and its different parts are described by block diagrams.

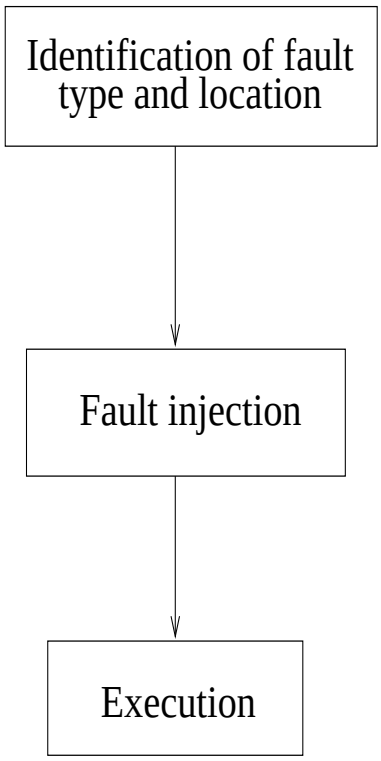


Figure 6.1: Simplified block diagram of the program realising fault injection

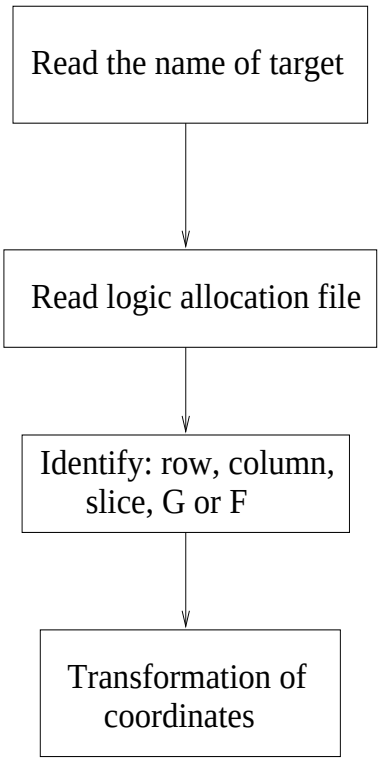


Figure 6.2: Block "Identification of block type and location" of fig. 6.1 in case of stuck-at fault injection

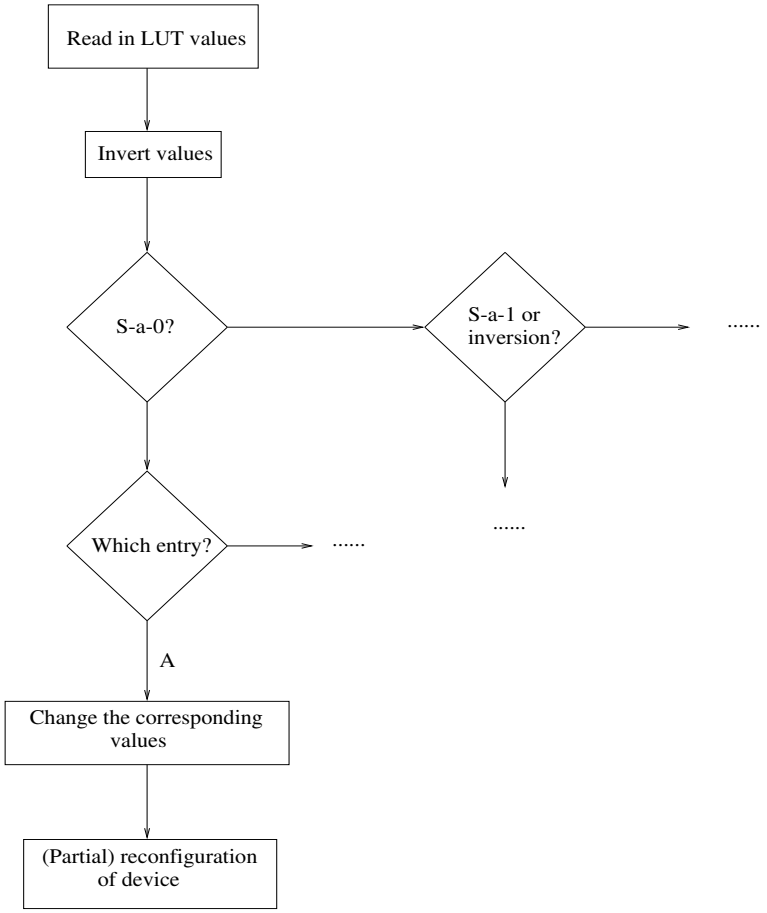


Figure 6.3: Block "Fault injection" of Fig. 6.1 in case of stuck-at fault injection

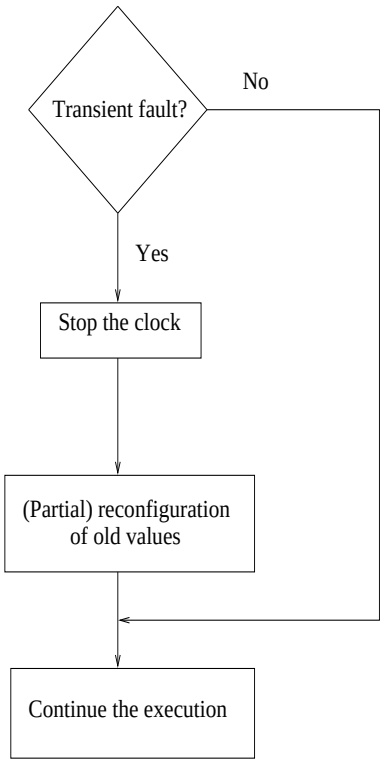


Figure 6.4: Block "Execution" of Fig. 6.1 in case of stuck-at fault injection

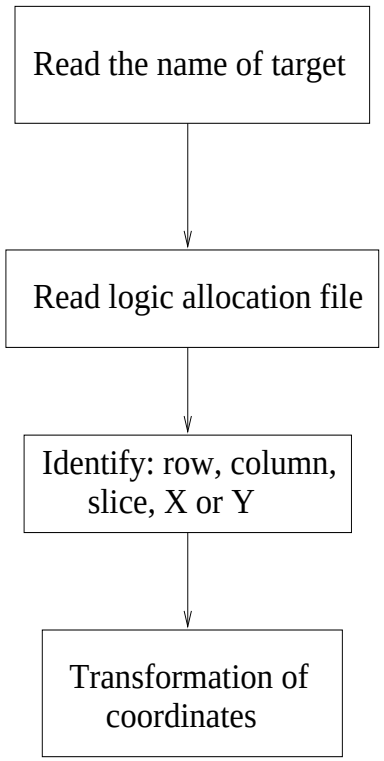


Figure 6.5: Block "Identification of block type and location" of Fig. 6.1 in case of SEU injection

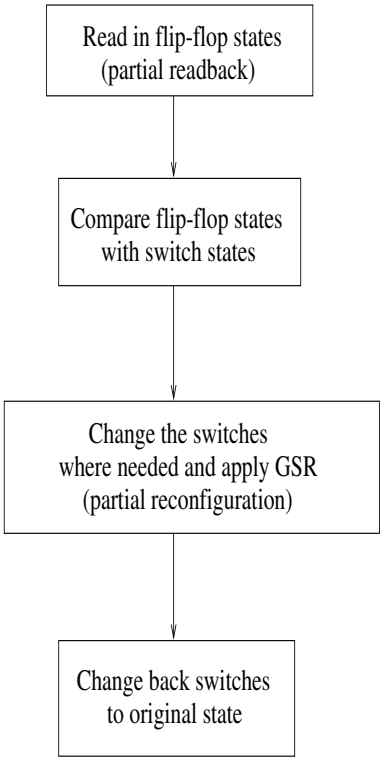


Figure 6.6: Block "Fault injection" of Fig. 6.1 in case of SEU injection

TITRE

Injection de fautes par reconfiguration dynamique de réseaux programmables

RESUME

Le travail présenté dans cette thèse entre dans le cadre des méthodes d'injection de fautes au niveau matériel, dans des circuits digitaux spécifiés dans un langage de haut niveau comme VHDL. Une nouvelle méthode est proposée et évaluée, basée sur l'utilisation de prototypes implémentés sur un réseau programmable reconfigurable. Les possibilités de reconfiguration partielle (ou locale) de certains de ces réseaux sont mises à profit pour réaliser directement l'injection des fautes, sans avoir à modifier la description VHDL du circuit sous test.
Cette thèse démontre la faisabilité d'une telle approche, pour deux types de fautes majeurs (les collages et les inversions de bits asynchrones, qui modélisent les fautes de type "Single Event Upset"). Les avantages et les limitations par rapport aux techniques existantes sont analysés, et les principaux paramètres devant être optimisés dans un tel environnement d'injection sont identifiés.

MOTS CLES

VLSI, conception sûre, analyse de sûreté, injection de fautes, reconfiguration dynamique

TITLE

Fault injection using run-time reconfiguration of FPGAs

ABSTRACT

The work presented in this thesis deals with hardware fault injection techniques in digital circuits specified in a high-level description language as VHDL. A new methodology is proposed and evaluated, based on prototypes implemented in reconfigurable hardware. Partial (or local) reconfiguration of programmable devices has been exploited for direct realisation of fault injection, without modifying the VHDL description of the circuit under test.
This thesis shows the feasibility of such an approach for two types of major faults (stuck-at faults and asynchronous bit-inversions modelling Single Event Upsets). The advantages and limitations comparing to the classical techniques are analysed, and the most important parameters that should be optimized in such a fault injection environment are identified.

KEYWORDS

key words : VLSI, dependable system design, dependability evaluation, fault injection, run-time reconfiguration

INTITULE ET ADRESSE DU LABORATORIE

Laboratoire TIMA, 46 avenue Félix VIALLET, 38031 GRENOBLE CEDEX, France

ISBN : 2-84813-014-8 **ISBNE : 2-84813-015-6**