Conception d'une architecture journalisée tolérante aux fautes pour un processeur à pile de données

Mohsin Amin

LABORATOIRE INTERFACES CAPTEURS
ET MICRO-ÉLECTRONIQUE

École Doctorale IAEM - Lorraine
Département de Formation Doctorale Électronique - Électrotechnique

THÈSE DE DOCTORAT

Présentée pour obtenir le grade de docteur de
l'Université Paul Verlaine - Metz

Discipline : Systèmes Électroniques
Spécialité : Microélectronique

CONCEPTION D'UNE ARCHITECTURE JOURNALISÉE TOLÉRANTE AUX FAUTES POUR UN PROCESSEUR À PILE DE DONNÉES

par

MOHSIN AMIN

Soutenue le 9 juin 2011 devant le jury composé de :

LUC HEBRARD	Pr., Université de Strasbourg	Examinateur
AHMED BOURIDANE	Pr., Université de Northumbria, Newcastle, Royaume Uni	Rapporteur
FERNANDO MORAES	Pr., Université PUCRS, Porto Alegre, Brésil	Rapporteur
CAMILLE DIOU	MCF, Université Paul Verlaine - Metz	Co-encadrant
FABRICE MONTEIRO	Pr., Université Paul Verlaine - Metz	Directeur de thèse

LICM - 7 Rue Marconi, Technopôle, 57070 Metz, France
Tel : +33 (0)3 87 31 56 57 - Fax : +33 (0)3 87 54 73 07 - www.licm.fr

Table des matières

INTRODUCTION GÉNÉRALE

Introduction Générale

Aujourd'hui, les dispositifs sont de plus en plus sensibles à l'impact des particules de haute énergie. Il y a de fortes probabilités qu'elles puissent causer des perturbations isolées (*Single Event Upset* – SEU) en frappant la surface du circuit silicium. Cela peut entraîner des erreurs temporaires (*soft errors*) qui se manifestent comme des inversions de bits dans la mémoire ou du bruit dans la logique combinatoire.

Ces dernières années, la performance des microprocesseurs a augmenté de façon exponentielle avec les tendances de conception modernes. Cependant, leur sensibilité aux effets de l'environnement s'est accrue [Kop11]. Alors que les fréquences d'horloge augmentent et que la taille des blocs fonctionnels diminue, ces systèmes peuvent devenir vulnérables aux rayonnements ionisants qui traversent l'atmosphère. En outre, les erreurs *soft* peuvent être déclenchées par des facteurs environnementaux tels que les décharges statiques ou les fluctuations de température et de tension d'alimentation. La survenue d'erreurs *soft* dans les systèmes électroniques modernes est en constante augmentation [Nic10]. La sûreté de fonctionnement est une préoccupation importante pour la conception des générations de processeurs actuelles et futures [RI08].

Les approches conventionnelles pour la conception de processeurs fiables exploitent la redondance spatiale ou temporelle [RR08]. La réplication du processeur a été utilisée pendant une longue période comme principale technique de tolérance aux fautes (*Fault Tolerance* – FT) contre les fautes transitoires [Kop04]. C'est une solution coûteuse, nécessitant plus de 100 % de surface et de puissance supplémentaires, car la duplication au minimum est nécessaire pour la détection des erreurs (triplication au minimum pour la correction ou le masquage d'erreurs) ainsi que des circuits voteurs supplémentaires. En pratique, c'est une solution coûteuse pour détecter les erreurs au niveau RTL, en particulier quand les SEU sont considérées. Les approches temporelles basées sur le logiciel ont un surcoût matériel moindre, et peuvent améliorer la sûreté de fonctionnement de manière significative [RI08]. Par exemple, dans le mode d'exécution duplex, toutes les instructions sont exécutées deux fois pour détecter les erreurs temporaires [MB07]. Cependant, cette technique a tendance à induire des surcoûts temporels significatifs, impliquant de sévères contraintes de temps difficiles à résoudre dans les circuits fonctionnant en temps réel. Ces approches peuvent fournir une tolérance aux fautes robuste mais encourir des pénalités élevées en termes de performances, de surface, et de puissance [RR08].

La redondance explicite est adaptée aux applications critiques où le coût du matériel n'est pas une contrainte importante. Toutefois, après une escalade technologique rapide, presque tous les systèmes

nécessitent aujourd'hui un minimum de tolérance aux fautes [FGAD10]. Ces systèmes exigent des solutions tolérance aux fautes moins onéreuses qui admettent un taux de couverture moindre que celui de la redondance matérielle, mais néanmoins important [RR08]. Par conséquent, des recherches sont nécessaires pour obtenir des solutions alternatives, non conventionnelles, et peu chères.

Nous proposons une nouvelle méthode de conception conjointe matérielle/logicielle pour la tolérance aux fautes transitoires dans les processeurs. La méthodologie repose sur deux choix principaux : détection d'erreur rapide et recouvrement d'erreur à faible coût. Une détection d'erreurs rapide est nécessaire afin que les erreurs puissent être détectées avant qu'elles ne se propagent au-delà des limites du système et provoquent des défaillances catastrophiques [1]. Par conséquent, la technique matérielle de détection concurrente d'erreurs (*Concurrent Error Detection* – CED) a été choisie. Pour contenir le coût global, nous pouvons accepter une légère pénalité temporelle lors de la correction d'erreur. Dans ce scénario, une méthode de restauration reposant sur le logiciel (*software based rollback*) est utilisée. Elle permettra de réduire le coût global par rapport à la restauration matérielle. Ainsi, les performances globales ne seront pas trop lourdement affectées, car cette méthode est adaptée aux applications au sol où l'occurence d'erreurs est nettement moindre que dans l'espace.

Nous considèrerons tout au long de ce travail qu'une mémoire sûre (*Dependable Memory* – DM) est attachée au processeur. Par ailleurs, pour rendre la restauration rapide et simplifier la gestion de la mémoire, nous utiliserons un stockage des données intermédiaire entre le processeur et la mémoire. Ici, les choix architecturaux sont importants pour que la méthode globale soit réussie. Par exemple, un cœur de processeur ayant un nombre d'états internes minimal à vérifier (lors de la détection d'erreur), ainsi qu'à charger et à sauvegarder (lors de la restauration) peut rendre cette technique efficace (moins chère et rapide). Le processeur tolérant aux fautes a été modélisé en VHDL au niveau RTL. Enfin, la capacité d'autocontrôle du processeur et la dégradation des performances due à la réexécution ont été testées par injection d'erreurs artificielle dans le modèle simulé.

Les contributions de ce travail sont les suivantes : Proposer une nouvelle méthodologie basée sur la conception conjointe matérielle/logicielle pour obtenir un bon compromis entre le niveau de protection et les contraintes de temps et de surface. Pour une détection d'erreur rapide, la détection concurrente matérielle est employée. Pour un faible surcoût matériel, un « micro-rollback » logiciel sera utilisé. Afin de réduire le surcoût en surface, nous emploierons un processeur à pile de la classe MISC (*Minimum Instruction Set Computer*). Le processeur a un nombre de registres internes minimal qui permet d'une part une détection d'erreur faible coût, et d'autre part en fait un choix adapté pour un recouvrement efficace des erreurs. De plus, pour empêcher les erreurs de se propager à la mémoire, un stockage temporaire intermédiaire est introduit entre le processeur et celle-ci.
Cette thèse est divisée en six chapitres.

Le *chapitre 1* donne un aperçu du contexte et décrit les motivations pour une détection d'erreurs en ligne et une correction rapide dans les microprocesseurs embarqués. Il présente les concepts de base et la terminologie liés à la sûreté de fonctionnement des processeurs embarqués. Il explore également

[1] où le coût des conséquences néfastes est de plusieurs ordres de grandeur – voire même incommensurablement – plus élevé que le bénéfice apporté par la prestation de service correcte [LRL04]

les attributs, les menaces, et les moyens permettant d'atteindre la sûreté de fonctionnement. Enfin, les différentes techniques de sûreté de fonctionnement appliquées à différents niveaux sont discutées.

Le *chapitre 2* présente les différentes techniques de redondance pour détecter et corriger les erreurs. Il explore les différentes méthodologies employées dans les processeurs tolérants aux fautes existants. La dernière partie est consacrée à la méthodologie de validation d'un processeur fiable.

Le *chapitre 3* identifie les spécifications du modèle et la méthodologie de conception de l'architecture désirée. Il traite du problème global en explorant le paradigme de conception et les contraintes liées à l'approche proposée. Plus tard, l'interface processeur-mémoire est finalisée par différentes implantations fonctionnelles en C++.

Le processeur proposé comporte deux parties : un cœur de processeur autocontrôlé (*Self-Checking Processor Core* – SCPC) et un journal matériel autocontrôlé (*Self-Checking Hardware Journal* – SCHJ). Le *chapitre 4* mène à une méthodologie de conception des SCPC. Le processeur a été choisi parmi les processeurs de classe MISC, par conséquent, nous éclaircissons tout d'abord les raisons du choix d'un tel processeur spécialisé. Ensuite, la détection d'erreur et le mécanisme de recouvrement sont finalisés. Enfin, le modèle matériel du cœur de SCPC est synthétisé sur un FPGA Stratix III d'Altera à l'aide de Quartus II.

Le *chapitre 5* traite de la conception matérielle et du système de protection du SCHJ, qui est un stockage de données temporaire permettant de masquer les erreurs et interdire leur propagation vers la mémoire. Enfin, le modèle matériel global du processeur est synthétisé sur un FPGA Stratix III d'Altera à l'aide de Quartus II.

Enfin, dans le *chapitre 6*, le modèle de processeur tolérant aux fautes est évalué en présence d'erreurs. L'évaluation est basée sur la dégradation de l'autocontrôle et des performances en présence d'erreurs. Par conséquent, les résultats obtenus valident les techniques de protection proposées dans le chapitre 3.

Enfin, la dernière section présente les conclusions et les perspectives.

I. État de l'art et étude théorique

Chapitre 1

Sûreté de fonctionnement et tolérance aux fautes

C'est une tâche complexe que de concevoir des systèmes embarqués pour les applications critiques en temps réel. Ces systèmes doivent non seulement garantir une réponse dans les délais imposés par leur environnement physique, mais également le faire de manière fiable, malgré l'occurrence de fautes [Pow10]. Le besoin en processeurs tolérants aux fautes s'est accru ces dernières années [Che08], et va probablement devenir la norme. Si, dans le passé, la tolérance aux fautes a été le domaine exclusif des applications très spécialisées comme les systèmes de sécurité critique, les tendances de conception moderne impliquent des circuits plus sensibles qui conduisent les systèmes temps réel à disposer d'au moins quelques fonctionnalités de tolérance aux fautes. Par conséquent, la tolérance aux fautes est un besoin important – et croissant – des nouvelles générations de systèmes.

Le système social moderne est articulé autour de l'industrie automatisée. Dans certains secteurs industriels sensibles, même une faute simple peut entraîner des pertes qui se chiffrent en millions dollars (par exemple dans les domaines bancaires ou boursiers), voire entraîner des pertes humaines (par exemple, les systèmes de contrôle du trafic aérien). Les industries comme l'automobile, l'avionique, et la production d'énergie nécessitent sûreté de fonctionnement, performance, et capacité de réponse en temps réel pour éviter les défaillances catastrophiques. Dans le tableau 1, le coût par heure des défaillances des systèmes de contrôle a été comparé afin d'illustrer l'importance et le besoin en sûreté de fonctionnement dans le secteur industriel.

TABLE 1.1 – Coût/heure des défaillances des systèmes de contrôle [Pie07]

Domaine d'application	Coût (€/h)
Opérateur de téléphonie mobile	40 k
Compagnies aériennes	90 k
Automates bancaires	2.5 M
Industrie automobile	6 M
Bourse des valeurs	6.5 M

La plupart des systèmes du tableau 1 s'appuient sur les systèmes embarqués. La conception du processeur tolérant aux fautes est l'une des exigences de base pour les applications embarquées fiables. En conséquence, nous proposons de concevoir un processeur afin d'éliminer (tolérer) les fautes transitoires qui résultent des SEU. Dans ce chapitre d'introduction, nous aborderons les notions de base et la terminologie liées à la tolérance aux fautes. Ce chapitre est divisé en trois parties principales : la première partie décrit les tendances actuelles qui augmentent la probabilité de fautes ainsi que les sources et conséquences des fautes. La deuxième partie examine le concept de processeur fiable et la troisième partie explore les moyens permettant d'atteindre la sûreté de fonctionnement.

1.1 Problématique

Pendant de nombreuses années, les chercheurs se sont focalisés sur les problèmes de performances. Grâce à leurs efforts incessants, ils ont ces dernières années amélioré la performance globale des processeurs, aidés en cela par le facteur technologique. Toutefois, les limites définies par la loi de Moore ont atteint leur niveau de saturation, et l'on observe une diminution de la sûreté de fonctionnement en raison de défauts physiques en constante augmentation. Différentes tendances dans la recherche de hautes performances ont accru le besoin en architectures fiables. Certaines d'entre-elles sont présentées ci-dessous :

Technologies plus fines / Échelle de conception

Bien que la réduction de la taille des transistors et des connexions n'a cessé d'augmenter les performances des processeurs et de réduire leur coût, elle a également affecté la sûreté de fonctionnement à long terme des puces. Quand un transistor est exposé à des rayonnements ionisants de haute énergie, des paires électron-trou sont créées [SSF+08]. La source du transistor et les zones de diffusion accumulent des charges qui peuvent inverser l'état logique du transistor [Muk08]. La finesse de gravure qui devrait être réduite à moins de 18 nanomètres en 2015 menace de manière significative les prochaines générations technologique [RYKO11]. Concernant les fautes transitoires, des circuits plus petits ont tendance à mettre en œuvre de plus faibles charges pour maintenir les états des registres, et les rendent de ce fait plus sensibles aux bruits. Lorsque la marge de bruit diminue, la probabilité que l'impact d'une particule de haute énergie puisse perturber la charge sur les circuits augmente, qui elle-même augmente à son tour la probabilité de fautes transitoires. De même, les tensions plus faibles mises en œuvre pour des raisons d'efficacité énergétique augmentera la sensibilité des puces à venir [FGAD10].

Plus de transistors par puce

Plus un circuit comporte de transistors, plus de fils sont nécessaires pour les connecter, résultant en une plus forte probabilité de fautes, tant lors de la fabrication que du fonctionnement de ces dispositifs. Les processeurs modernes sont plus sujets aux fautes en raison d'un plus grand nombre de transistors

et de registres. Par ailleurs, la température est un autre facteur de fautes transitoires ou permanentes. Plus la puce comporte de dispositifs, plus de puissance sera drainée de l'alimentation, ce qui – à surface égale – va augmenter les pertes par dissipation avec comme conséquence une augmentation de la température et de la probabilité d'erreurs.

Architectures plus complexes

Aujourd'hui, les processeurs sont plus compliqués comparés aux architectures passées, ce qui augmente la probabilité d'erreurs de conception. D'autre part, cela rend également difficile le débogage des erreurs. Les efforts de recherche sont orientés vers des méthodes alternatives pour augmenter les performances du système sans augmenter la sensibilité du circuit, mais malheureusement le goulot d'étranglement a été atteint et ces solutions plus complexes font du débogage une tâche plus difficile à accomplir.

En résumé, les appareils sont plus en plus sensibles aux rayonnements ionisants (qui peuvent causer des erreurs temporaires, dites « *softs* »), aux variations du point de fonctionnement dues aux fluctuations de la tension d'alimentation ou de la température, ainsi qu'aux effets parasites qui entraînent des courants de fuite statique [ITR07]. Changer les paramètres comme les dimensions, la marge de bruit ou la tension d'alimentation ne peut plus permettre d'augmenter les performances.

Dans un proche avenir, à cause de leur petite taille et de leur fréquence élevée, la tendance à la défaillance des processeurs modernes sera en augmentation parce que le niveau de saturation est déjà atteint, conduisant à la hausse du taux d'erreurs *softs* dans les circuits logiques et les mémoires [Bau05], ce qui en affecte la sûreté de fonctionnement, même au niveau de la mer [WA08]. Pour assurer l'intégrité du circuit, la tolérance aux fautes doit être une considération importante dans la conception des circuits modernes. Un système fiable doit être conscient du mécanisme de tolérance contre les possibles erreurs.

1.1.1 Sources communes de fautes et leurs conséquences

Aujourd'hui, une menace importante pour la sûreté de fonctionnement des circuits numériques concerne la sensibilité des états logiques aux diverses sources de bruit et spécifiquement dans certains milieux tels que dans l'espace ou le systèmes nucléaires, où les collisions de particules chargées peuvent entraîner des fautes transitoires. Ces particules peuvent inclure les rayons cosmiques produits par le soleil et les particules alpha produites par la désintégration d'isotopes radioactifs.

Pour les applications spatiales, la tolérance aux fautes est une exigence en raison de l'environnement radiatif sévère. Comme la technologie de fabrication évolue vers des géométries de plus en plus fines, la probabilité de SEU est en augmentation. Avec les technologies actuelles, la sûreté de fonctionnement n'est pas seulement nécessaire pour certaines applications critiques : même pour les systèmes grand public, la sûreté de fonctionnement doit être au-dessus d'un certain niveau pour que le système soit utile à quoi que ce soit [FGAD10]. Les erreurs *softs* induites par les radiations sont une menace de plus en plus importante pour la sûreté de fonctionnement des circuits numériques, même

dans les applications au niveau du sol [Nic10].

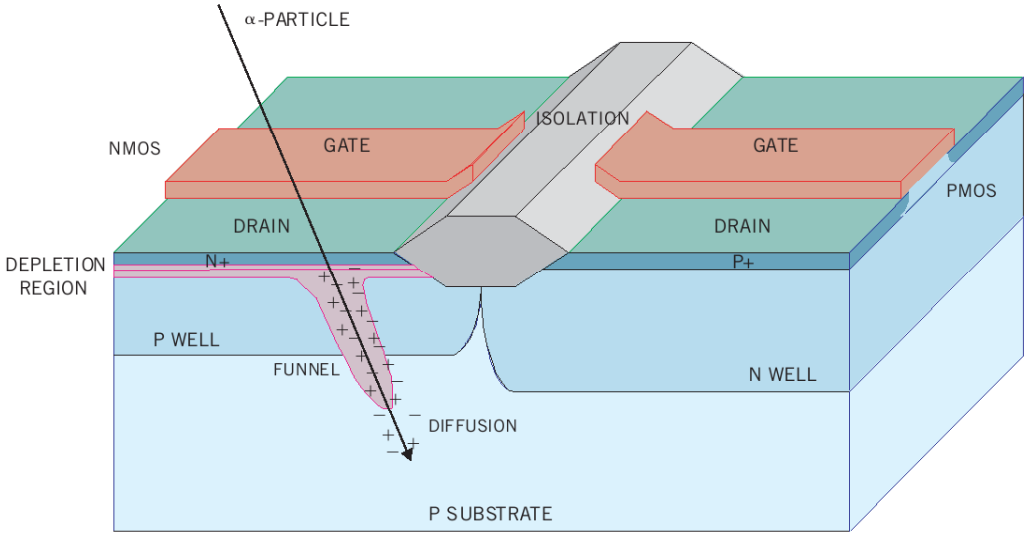


FIGURE 1.1 – Une particule alpha heurte un transistor CMOS. La particule génère des paires électron-trou dans son sillage, ce qui génère des perturbations de charge [MW04]

Les fautes transitoires peuvent être causées par des perturbations sur la puce, comme du bruit d'alimentation ou du bruit externe [NX06]. Les chercheurs ont répertorié trois principales sources d'erreurs *softs* dans les semi-conducteurs, dont la particule alpha, découverte en avant, s'est avéré être la principale source d'erreurs *softs* dans les systèmes de calcul, et particulièrement la DRAM [MW07]. Deuxièmement, les neutrons de haute énergie provenant des radiations cosmiques pourraient induire des erreurs *softs* dans les dispositifs semi-conducteurs via les ions secondaires produits par la réaction des neutrons avec les noyaux de silicium [ZL09] comme indiqué dans la figure 1.1, où un seul neutron de haute énergie a perturbé la distribution interne des charges de l'ensemble du circuit. Troisièmement, une source d'erreurs interne est induite par des interactions entre des neutrons cosmiques de faible énergie et l'isotope bore-10 dans les matériaux des circuits intégrés, en particulier dans le BPSG (*BoroPhosphoSilicate Glass*), largement utilisé pour former des couches d'isolant dans la fabrication de circuits intégrés. Ceci a récemment été identifié comme la source dominante d'erreurs *softs* dans les mémoires SRAM fabriquées avec le BPSG [WA08].

La figure 1.2 représente la séquence des événements qui peuvent survenir dès qu'une particule énergétique frappe le substrat, provoquant son ionisation. Cette ionisation peut générer un ensemble de paires électron-trou qui créent un courant transitoire qui est injecté dans – ou extrait de – ce nœud. Selon l'amplitude et la durée de cette impulsion en courant, une impulsion en tension transitoire peut apparaître au niveau du nœud frappé. Ceci est caractérisé comme la faute. Il existe une période de latence de faute qui définit le temps nécessaire pour que la faute se manifeste en erreur dans le circuit. Cela ne se produira que si la tension transitoire de ce nœud change la logique d'un élément de stockage (bascule, ou *flip-flop*), générant une inversion de bit (*bit-flip*). Ce *bit-flip* peut générer une erreur si le contenu de cette bascule est utilisé pour une certaine opération. Toutefois, du point

de vue de l'application, il n'est pas obligatoire que cette erreur se manifeste en défaillance dans le système. Il existe aussi une latence d'erreur qui définit le temps nécessaire pour que l'erreur devienne une défaillance dans le système. Le terme couramment employé pour désigner un effet mesurable

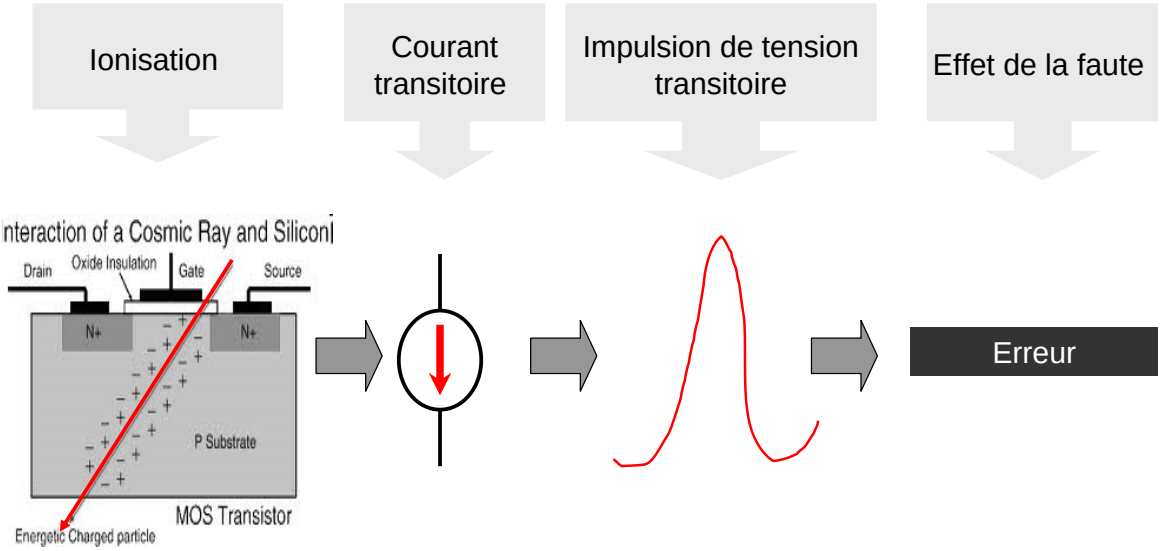


FIGURE 1.2 – Impact d'une particule de haute énergie entraînant une erreur

résultant du dépôt d'énergie par l'impact d'une unique particule ionisante est SEE (*Single Event Effect*). Les SEE les plus pertinents sont répertoriés dans la figure 1.3.

Single Event Upset (SEU)

Le SEU est la plupart du temps une erreur temporaire causée par le signal transitoire induit par l'impact d'une particule isolée de haute énergie [JES06]. Dans [Bau05], il est précisé qu'il se produit lorsqu'une rayonnement provoque une perturbation de charge suffisamment importante pour inverser l'état d'une cellule mémoire, registre, verrou, ou *flip-flop*. L'erreur est dite « *soft* », car le circuit n'est pas endommagé de façon permanente par le rayonnement, et lorsqu'une nouvelle donnée est écrite dans la cellule mémoire touchée, le dispositif va la stocker correctement [Bau05].

Le SEU est un problème très grave car il est l'une des principales sources de défaillance dans les systèmes numériques [Nic10]. Il constituera probablement une menace sérieuse pour l'avenir du calcul robuste [RK09] et exige une attention sérieuse. Il peut se manifester comme SBU (*Single Bit Upset*) ou MBU (*Multi Bit Upset*).

Single Bit Upset (SBU) et *Multi Bit Upset* (MBU)

Le SBU est un événement unique dû à un rayonnement qui se traduit par une inversion de bit alors qu'un MBU est un événement unique dû à un rayonnement qui résulte en ce que plus d'un bit soient inversés. Chaque inversion de bit est par essence un SEU. Les SBU et MBU sont donc considérés comme un sous-ensemble du SEU. Les SBU constituent habituellement la majeure partie, et les

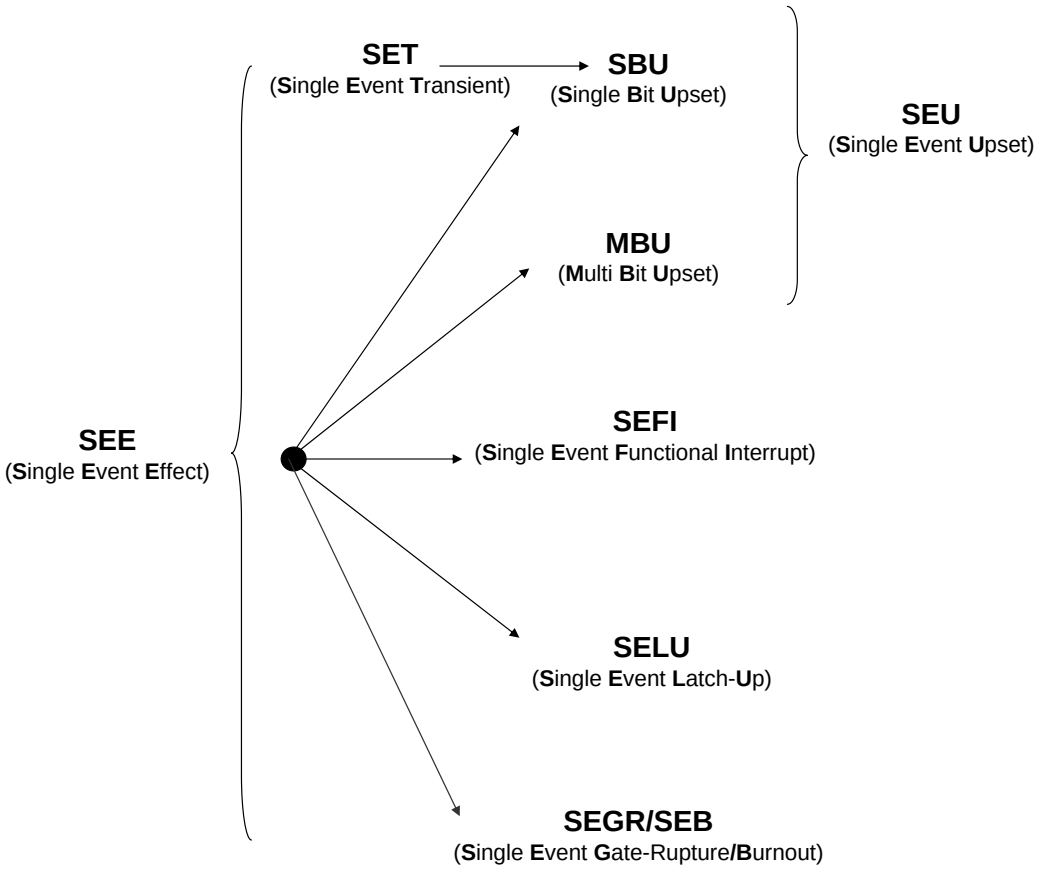


FIGURE 1.3 – Classification des fautes dues aux SEE (*Single Event Effects*) [Pie07].

MBU ne représentent qu'une petite part du nombre total de SEU observés. Cependant, la probabilité d'occurrence des MBU est en constante augmentation au fur et à mesure que la gravure gagne en finesse [BCT08, QGK+06]. Actuellement, cette thèse traite des SBU. À l'avenir, la méthodologie sera étendue pour traiter les MBU.

Single Event Transient (SET)

Le SET est une impulsion transitoire dans le chemin logique d'un circuit intégré. Semblable à un SEU, il est induit par un dépôt de charge d'une particule ionisante isolée. Un SET peut être propagé le long du chemin logique où il a été créé. Il peut être verrouillé dans un registre, un verrou ou une *flip-flop*, provoquant le changement de leur état de sortie.

Single Event Functional Interrupt (SEFI)

Xilinx [BCT08] définit un SEFI comme un SEE qui interfère avec le fonctionnement normal d'un circuit numérique complexe. De même que pour le SET précédemment mentionné, une enquête plus poussée des taux de SEFI ne sera pas effectuée dans cette thèse.

***Single Event Latch-Up* (SELU)**

Un pic de courant induit par une particule ionisante dans un transistor peut être amplifié par l'importante rétroaction positive du thyristor et provoquer un court-circuit virtuel entre Vdd et la masse, résultant en un SELU [NTN+09]. Les SELU ne sont pas considérés dans cette thèse.

***Single Event Gate Rupture* (SEGR) et *Single Event Burnout* (SEB)**

Le SEGR est l'état induit par un ion isolé dans des MOSFET de puissance qui peut entraîner la formation d'un chemin conducteur dans l'oxyde de grille. Le SEB est une condition qui peut causer la destruction du circuit par un état de courant élevé dans un transistor de puissance. Les deux sont des fautes permanentes et ne seront donc pas abordés dans cette thèse.

1.2 Concepts de base et taxonomie du calcul sûr

Cette partie définit la terminologie de base liée à la sûreté de fonctionnement des processeurs. La terminologie est principalement extraite de [LB07, Lap04]. Dans cette section, nous identifions les principales méthodes et leurs caractéristiques pour rendre un système tolérant aux fautes.

1.2.1 Sûreté de fonctionnement

La sûreté de fonctionnement est la capacité à offrir un service auquel on peut légitimement faire confiance [Lap04]. La définition est axée sur la confiance. En d'autres termes, la sûreté de fonctionnement d'un système est sa capacité à éviter les défaillances de service qui sont plus fréquentes et plus graves que ce qui peut être toléré. Elle s'appuie sur un ensemble de mesures qui, pendant toutes les étapes de la vie du produit, permettent de s'assurer que la fonctionnalité sera maintenue tout en accomplissant la mission pour laquelle il a été conçu. Selon Laprie [LB07], la sûreté de fonctionnement d'un système est la propriété de lui accorder une confiance justifiée dans le service qu'il fournit.

1.3 Attributs

La sûreté de fonctionnement est un concept vaste basé sur des attributs différents, comme indiqué dans la figure 1.4.

- *Disponibilité* : c'est le fait d'être prêt à accomplir un service correct ;
- *Fiabilité* : c'est la continuité du service correct ;
- *Sécurité-innocuité* : c'est l'absence de conséquences catastrophiques pour l'utilisateur ou l'environnement ;

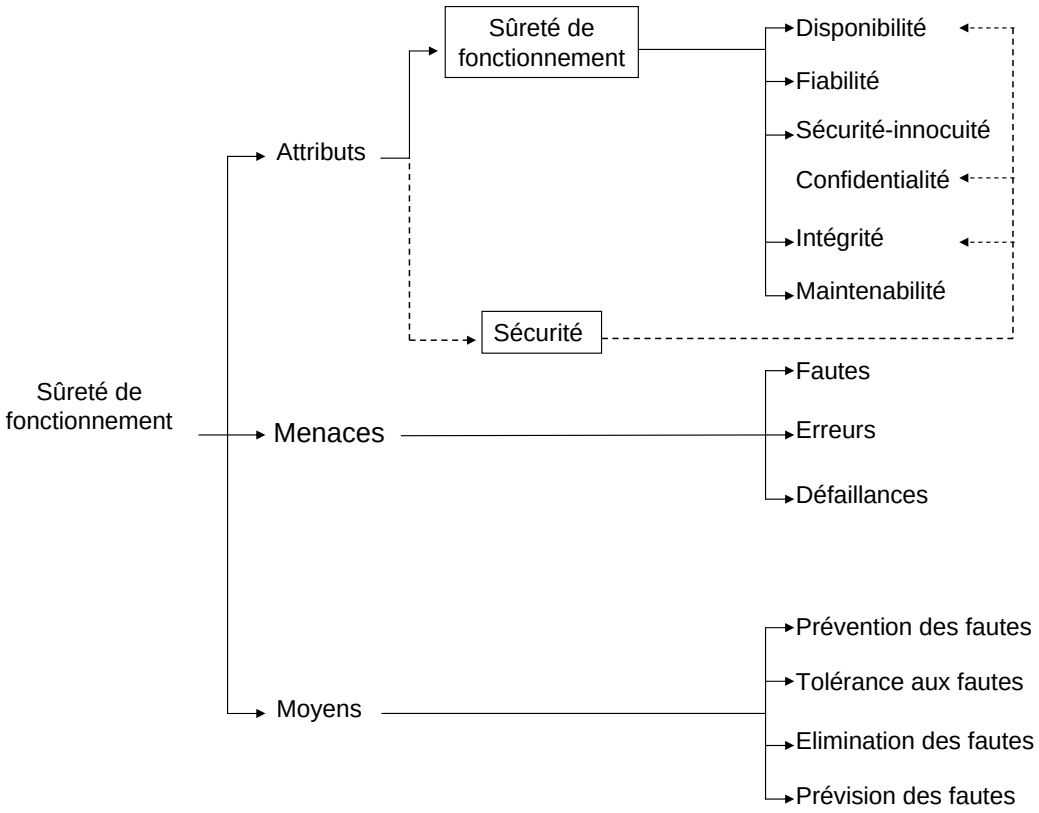


FIGURE 1.4 – Arbre de la sûreté de fonctionnement

- *Intégrité* : c'est l'absence d'altérations inadéquates du système ;
- *Maintenabilité* : c'est la capacité à subir des modifications.

En outre, lorsque l'on traite des questions de sécurité, un attribut supplémentaire appelé « confidentialité » est également considéré comme le montre la figure 1.4. La confidentialité est l'absence de divulgation non-autorisée de l'information.

Il est difficile de respecter tous les attributs de la sûreté de fonctionnement à la fois dans un système, car cela peut augmenter le coût, la consommation d'énergie et la surface du système. Ainsi, on respecte ces attributs selon les besoins du système. Il a été prouvé dans [FGAM10] qu'il est impossible de concevoir un système de 100 % fiable. Par exemple, dans le but d'améliorer la disponibilité des composants, parfois on néglige la maintenance et la sécurité diminue en conséquence. Dans l'exemple suivant, deux types de systèmes ont été considérés :

- un serveur web
- un réacteur nucléaire

Voyons quels attributs de sûreté de fonctionnement et de sécurité sont les plus importants pour chacun de ces systèmes. Dans un serveur web d'université, la disponibilité est l'attribut important parce que chaque étudiant a besoin d'y accéder régulièrement, alors que pour un réacteur nucléaire,

les attributs comme la disponibilité, la fiabilité, la sécurité et la maintenabilité sont des considérations importantes. [Pie07] résume l'importance de ces attributs ; dans la table 1.2, 4 points ont été attribués aux attributs très importants et 1 point pour l'attribut le moins important. Ainsi, le tableau 1.2 chaque application a ses propres exigences de sûreté de fonctionnement et de sécurité.

TABLE 1.2 – Attributs de sûreté de fonctionnement pour un serveur web et un réacteur nucléaire [Pie07], où les attributs sont répertoriés comme : – très important = 4 points, – moins important = 1 point

Attribut	serveur web	réacteur nucléaire
Disponibilité	3	4
Fiabilité	1	4
Sécurité	1	4
Confidentialité	2	1
Intégrité	2	3
Maintenabilité	2	4

1.4 Menaces

Il ya trois menaces fondamentales pour un processeur fiable. Elles sont : (i) faute, (ii) erreur (iii) défaillance. La faute est définie comme un état erroné du matériel ou du logiciel résultant d'une défaillance du composant, des interférences physiques de l'environnement, d'une erreur de l'opérateur, ou d'une conception erronée [Pie06]. Une faute est active quand elle produit une erreur, sinon elle est considérée comme une faute dormante. Une faute active peut être une faute interne qui a été préalablement dormante. L'erreur est elle-même causée par une faute et une défaillance survient lorsqu'il y a déviation du service correct en raison d'une erreur. Toutes trois ont une incidence sur la cause et la relation entre elles, comme le montre la figure 1.5. En général, les fautes actives causent une erreur. Elles peuvent se propager d'un endroit à un autre à l'intérieur du système. Dans la figure 1.6, une erreur produite dans le processeur a été transférée à la mémoire principale. Par ailleurs, si une erreur atteint les limites du système, elle peut entraîner la défaillance du système, forçant le service fourni à s'écarter de sa spécification [GMT08] (voir figure 1.5). Si le système initial est un sous-système d'un système global alors cela peut provoquer une faille dans le système global. De cette manière, la chaîne faute–erreur–défaillance continuer de progresser.

Un SEU peut entraîner une défaillance du système, comme dans la figure 1.7 : l'impact d'un neutron de haute énergie (dû aux rayons cosmiques) sur un circuit VLSI a généré un SBU, qui a provoqué une erreur dans le système de contrôle du trafic et a finalement abouti à la défaillance du système.

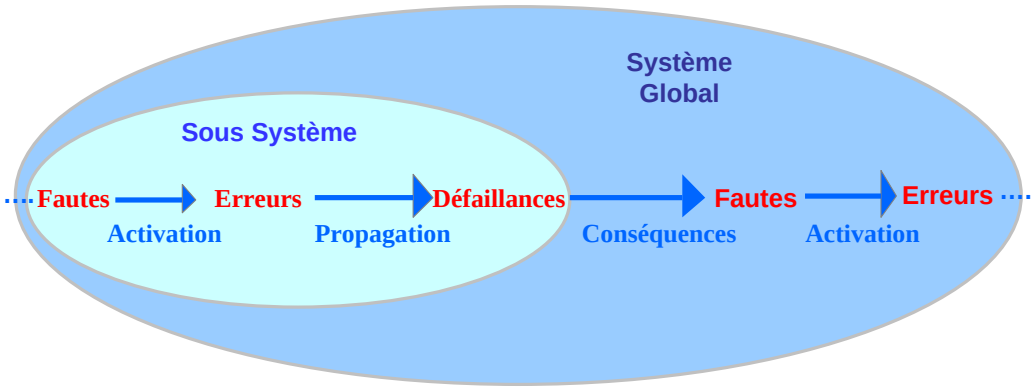


FIGURE 1.5 – Chaîne de faute, erreur et défaillance

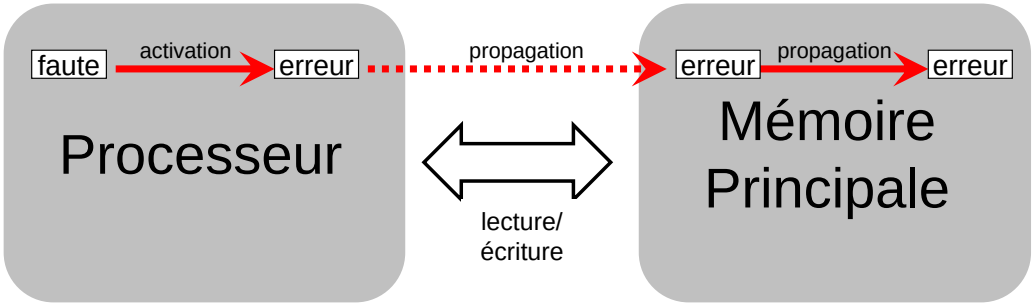


FIGURE 1.6 – Propagation d'erreur du processeur à la mémoire principale

1.4.1 Défaillance du système

Un service correct est délivré par un système lorsqu'il respecte sa fonctionnalité, alors qu'une défaillance du système est une déviation de sa spécification du service rendu par le système [Pie06]. Un tel écart peut prendre la forme d'un service incorrect, ou pas de service de tout [GMT08]. La transition d'un service incorrect vers un service correct est appelée « restauration de service » (voir figure 1.8).

La défaillance du service peut se produire parce que le système ne respecte plus sa fonctionnalité, ou parce que les spécifications fonctionnelles n'ont pas été correctement définies pour ce système sous certaines conditions. D'autre part, les techniques de tolérance aux fautes permettent à un système de fournir en permanence son service en accord avec sa fonctionnalité correcte, même en présence de fautes.

1.4.2 Les caractéristiques des fautes

Les fautes peuvent être caractérisées par les cinq attributs que sont la cause, la nature, la durée, l'étendue, et la valeur. La figure 1.9 illustre chacune de ces caractéristiques de base des fautes. Elles sont présentées dans la section suivante.

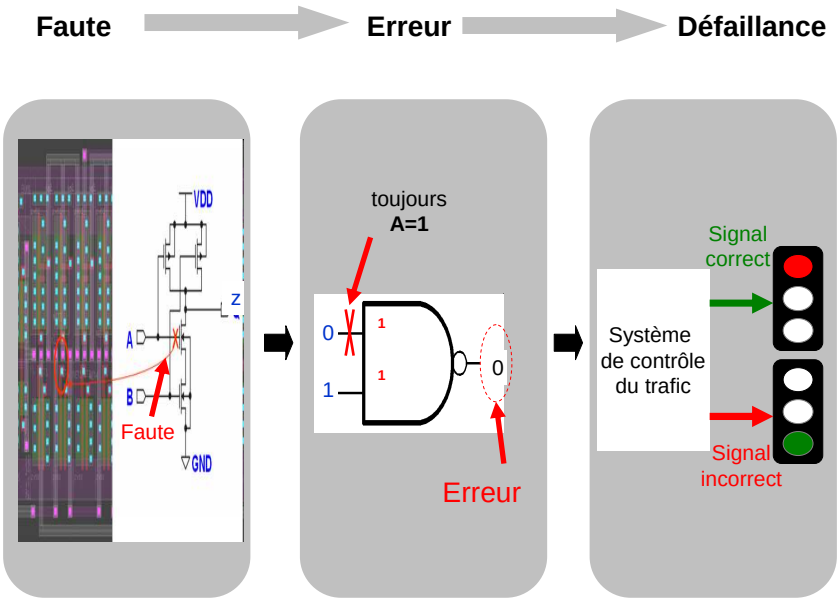


FIGURE 1.7 – Une seule faute a causé la défaillance du système de contrôle du trafic

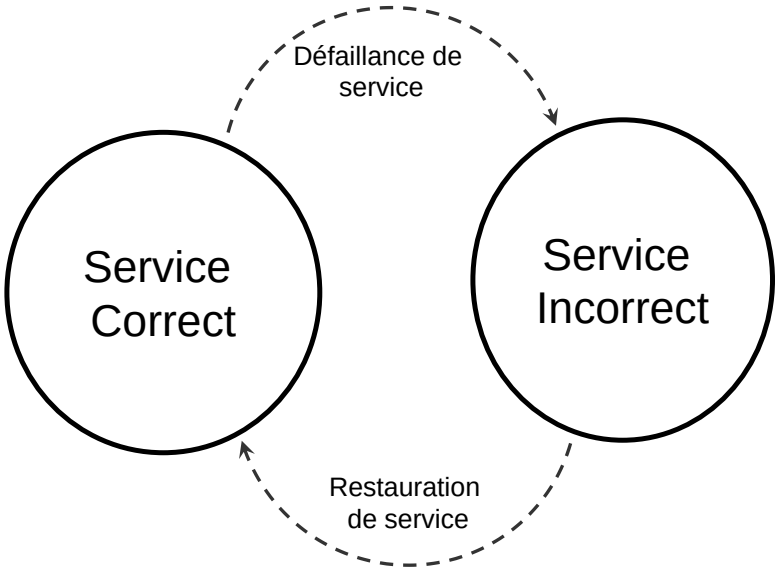


FIGURE 1.8 – Défaillance du service

Cause

Les fautes peuvent être dues à quatre problèmes importants :

1. *Erreurs de spécification :* Il s'agit notamment d'algorithmes, d'architectures, ou de spécifications incorrects, comme dans la ligne 1 de la figure 1.10, où il y a une faute causée par la mauvaise interconnexion entre les deux systèmes.

2. *Erreurs d'implantation :* L'implantation peut introduire des fautes dues à une mauvaise conception, une mauvaise sélection des composants, une mauvaise construction, ou des erreurs de co-

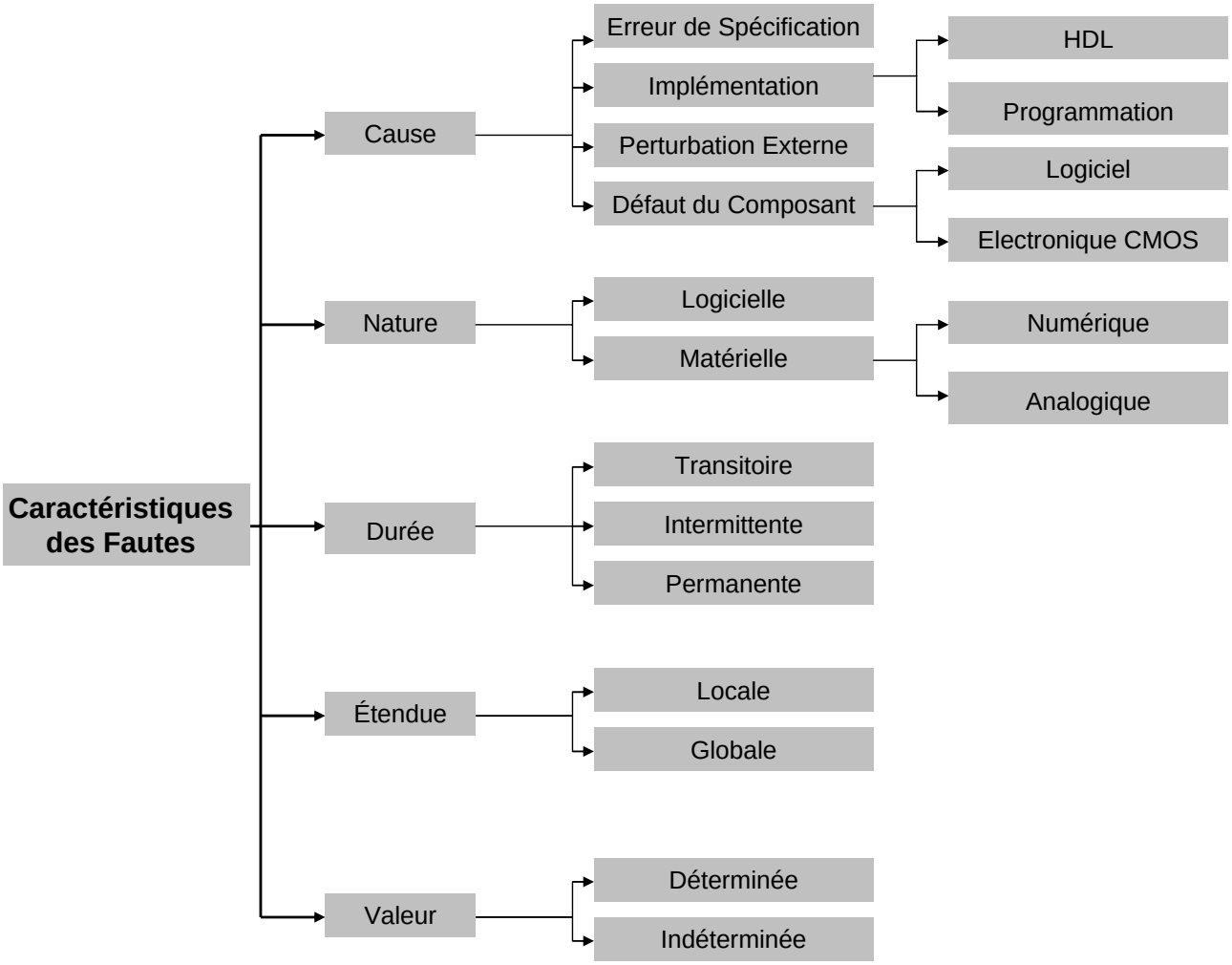


FIGURE 1.9 – Les caractéristiques d'une faute.

dage matériel ou logiciel comme dans les lignes 2 et 3 de la figure 1.10. La ligne 2 montre une faute de programmation où `c` est incrémenté si `a` est inférieur à `b` mais `c` ne sera pas incrémenté si `a` est égal à `b`, ce qui est une erreur de programmation. De même, dans la ligne 3 `r1` charge le résultat de l'addition `a + b` dans le registre `c`.

3. *Composants défectueux :* Cela inclut les dispositifs défectueux, les imperfections de fabrication, et l'usure des composants. Ça peut être un composant logique ou de l'électronique CMOS, comme montré dans les lignes 4 et 5 de la figure 1.10.

4. *Perturbations externes :* Cela inclut les erreurs de l'opérateur, le rayonnement, les interférences électromagnétiques, et les environnements extrêmes, comme dans la ligne 6 de la figure 1.10. Par ailleurs, en raison de la réduction de la marge de bruit, « 1 » peut être lu comme « 0 » si sa valeur est inférieure au seuil Vm, comme indiqué dans la ligne 7 de la figure 1.10).

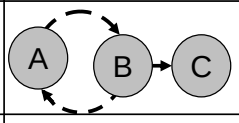
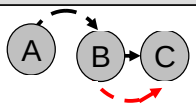
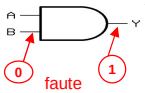
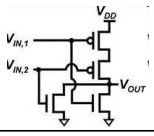
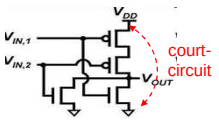
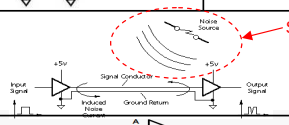
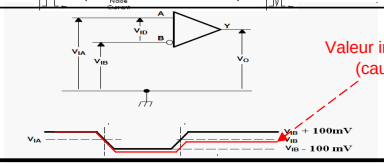
Sr. No.	Fautes à différents niveaux	État correct	État incorrect
1	Erreurs de spécification		
2	Faute de programmation	`if a < = b c : c + 1; end;`	`if a < b c : c + 1; end;`
3	HDL	`r1 : c <= a + b`	`r1 : c <= a + b` 
4	Défauts des composants 1		
5	Défauts des composants 2		
6	Perturbation externe		
7	Faute due à la marge de bruit inférieur		

FIGURE 1.10 – Quelques raisons d'apparition de fautes.

Nature

La nature d'une faute spécifie le type de faute, qui peut être matérielle ou logicielle. Dans le cas des fautes matérielles, elle en spécifie la nature analogique ou numérique.

Durée

La durée d'une faute spécifie la longueur de temps pendant laquelle la faute est active. Les fautes ont été classées en trois types en fonction de leur durée : transitoire, intermittente, et permanente.

- *Transitoire :* Une faute transitoire apparaît une fois, et ne continue pas. Une erreur provoquée par une telle faute est souvent désignée comme une erreur *soft*. De telles erreurs peuvent conduire à des données corrompues, l'exécution incorrecte du programme, ou une complète perturbation d'un programme en cours [Bic10].

- *Intermittent :* C'est une faute qui apparaît, disparaît, et réapparaît à plusieurs reprises dans un délai très court. Une défaillance intermittente peut se produire à plusieurs reprises, mais pas de façon continue pendant une longue période dans un dispositif.

- *Permanente :* Il s'agit d'une faute qui continue d'exister indéfiniment si aucune mesure corrective n'est prise. Une faute permanente, lorsqu'elle survient une fois, persiste jusqu'à la fin de l'exécution. Même une seule faute permanente peut créer de multiples erreurs jusqu'à ce qu'elle soit réparée. Ces erreurs sont appelées erreurs matérielles.

Les erreurs dans les processeurs modernes peuvent survenir en raison de fautes permanentes, intermittentes, ou transitoires. Cependant, les fautes transitoires surviennent beaucoup plus souvent que les permanentes, et sont beaucoup plus difficile à détecter [RS09]. Le ratio des fautes transitoires sur permanentes peut varier entre 2 :1 et 100 :1, voire plus. Ce ratio est en constante augmentation [Kop04].

Étendue

L'étendue d'une faute précise si la faute est localisée à un bloc matériel ou un module logiciel donnés, ou si elle affecte globalement le matériel, le logiciel, ou les deux.

Valeur

La valeur d'une faute peut être déterminée ou indéterminée. Une faute déterminée est celle dont l'état reste inchangé au long du temps, sauf s'il y a une action extérieure sur elle, alors qu'une faute indéterminée est une faute dont l'état à certains instants t peut être différent de son état à un autre moment.

1.5 Moyens

Il y a quatre moyens pour atteindre la sûreté de fonctionnement : la prévention des fautes, la tolérance aux fautes, l'élimination des fautes et la prévision des fautes. La prévention des fautes et la tolérance aux fautes ont pour objectif de fournir la capacité à délivrer un service auquel on peut se fier, alors que l'élimination et la prévision de faute ont pour but d'atteindre la confiance en cette capacité en justifiant que les spécifications fonctionnelles, de sûreté de fonctionnement et de sécurité sont adaptées, et que le système est susceptible de les atteindre [LRL04, LB07].

1.5.1 Prévention des fautes

La prévention des fautes est la capacité à éviter l'apparition ou l'introduction de fautes. Elle comprend toute technique qui tente de prévenir l'apparition de fautes. Il peut s'agir de vérification du design, de l'inspection des composants, du test, ou d'autres méthodes de contrôle de la qualité.

1.5.2 Élimination des fautes

L'élimination des fautes est la capacité à réduire le nombre et la gravité des fautes. Elle peut être effectuée durant les processus de maintenance corrective ou préventive. La maintenance corrective

vise à éliminer les fautes qui ont déjà produit une erreur et commence après la détection de l'erreur, tandis que l'entretien préventif est destiné à éliminer les fautes avant qu'elles ne puissent causer des erreurs [LB07].

1.5.3 Prévision des fautes

C'est la capacité à estimer le nombre actuel, l'incidence future, et les conséquences probables des fautes. Elle est réalisée en effectuant une évaluation du comportement du système à l'égard de la survenance ou de l'activation de fautes ; elle a deux aspects qui sont d'ordre qualitatif et quantitatif. Les principales approches probabilistes de prévision de défaillance visant à dériver des estimations probabilistes sont la modélisation et le test [LB07].

1.5.4 Tolérance aux fautes

La tolérance aux fautes est définie comme la capacité de corriger la fonctionnalité du système en présence de fautes. Idéalement, un système tolérant aux fautes est capable d'exécuter sa tâche correctement indépendamment de la présence de fautes. Cependant, en pratique personne ne peut garantir la parfaite exécution des tâches en toutes circonstances. Les systèmes tolérants aux fautes réels sont conçus pour être tolérants aux fautes les plus probables. Dans ce travail, la tolérance aux fautes a été considérée. Elle repose sur trois piliers que sont le masquage de fautes, la détection d'erreurs, et de la correction et le recouvrement d'erreurs.

Masquage de fautes

Le masquage de fautes cache les effets de fautes à l'aide de méthodes où l'information redondante l'emporte sur les informations erronées [Pie06]. Il s'agit d'une technique de redondance structurelle qui masque complètement les fautes dans les modules du système redondant. Un certain nombre de modules identiques exécutent les mêmes fonctions, et un vote en sortie permet de supprimer les erreurs créées par un module défectueux ; par exemple, la TMR (*Triple Modular Redundancy*) est une technique couramment utilisée de masquage de fautes.

Grâce au masquage de fautes, nous atteignons la sûreté de fonctionnement en cachant les fautes qui surviennent. Il empêche les effets des fautes de se répandre dans tout le système. Il peut tolérer les fautes du logiciel et du matériel comme indiqué dans la figure 1.11. Un tel système n'a pas besoin de détection et de correction d'erreur pour maintenir la sûreté de fonctionnement. Le masquage de fautes n'a pas été directement utilisé dans cette thèse. Toutefois, la TMR sera utilisée à titre de comparaison dans les chapitres suivants.

Détection d'erreur

Si le masquage n'est pas utilisé, alors la détection d'erreurs peut être employée dans un système tolérant aux fautes. La détection des erreurs est la pierre angulaire d'un tel système, car un système ne peut tolérer une erreur s'il n'a pas connaissance de son existence. Les mécanismes de détection d'erreur forment la base d'un système résistant aux erreurs car toute faute lors d'une opération doit être détectée avant que le système ne puisse décider d'une action corrective pour la tolérer [LBS+11]. Même si un système ne peut pas recouvrir l'erreur détectée, il peut au minimum stopper le processus ou informer l'utilisateur qu'une erreur est détectée et que les résultats ne sont plus fiables.

Correction et recouvrement d'erreurs

Détecter une erreur est une action suffisante pour assurer la sécurité, mais nous souhaitons également que le système puisse recouvrer les états défectueux. Le recouvrement cache les effets de l'erreur à l'utilisateur. Après le recouvrement, le système peut reprendre son fonctionnement et, idéalement, continuer à fonctionner normalement. Le recouvrement d'erreurs est une caractéristique importante pour un système basé sur les deux attributs de sûreté de fonctionnement et de disponibilité, car les deux mesures nécessitent que le système puisse se remettre de ses erreurs, sans intervention de l'utilisateur.

La détection d'erreur et le recouvrement sont abordés dans cette thèse, ils seront discutés en détail dans le chapitre 2. De même, différentes techniques de détection d'erreur (dans la section 2.1) et la correction (dans la section 2.2) seront également traités.

1.6 Techniques appliquées à différents niveaux

La figure 1.11 illustre les techniques de sûreté de fonctionnement appliquées à différents niveaux dans un système matériel et logiciel dans lequel l'évitement de fautes (prévention des fautes) est la principale méthode pour améliorer la sûreté de fonctionnement du système. Il peut être pris en compte par le biais du matériel ou d'implantations logicielles. L'évitement de faute dans un système matériel peut être atteint par la prévention des fautes de spécification et d'implantation, du défaut des composants, et des perturbations externes, alors que dans un système logiciel il nécessite la prévention des fautes de spécification et d'implantation. D'autre part, le masquage de fautes est une technique utilisée pour assurer la sûreté de fonctionnement, en masquant les fautes. La TMR est un exemple bien connu de cette technique. Si le masquage de faute n'est pas appliqué, alors la tolérance aux fautes est un choix adapté pour surmonter les erreurs.

1.6.1 Techniques de tolérance aux fautes

Les techniques de tolérance aux fautes pour les circuits intégrés peuvent être appliquées à différents moments lors du flot de conception. Elles peuvent être appliquées dans la phase de conception électrique, comme le dimensionnement de transistors, la redondance des transistors ou en ajoutant

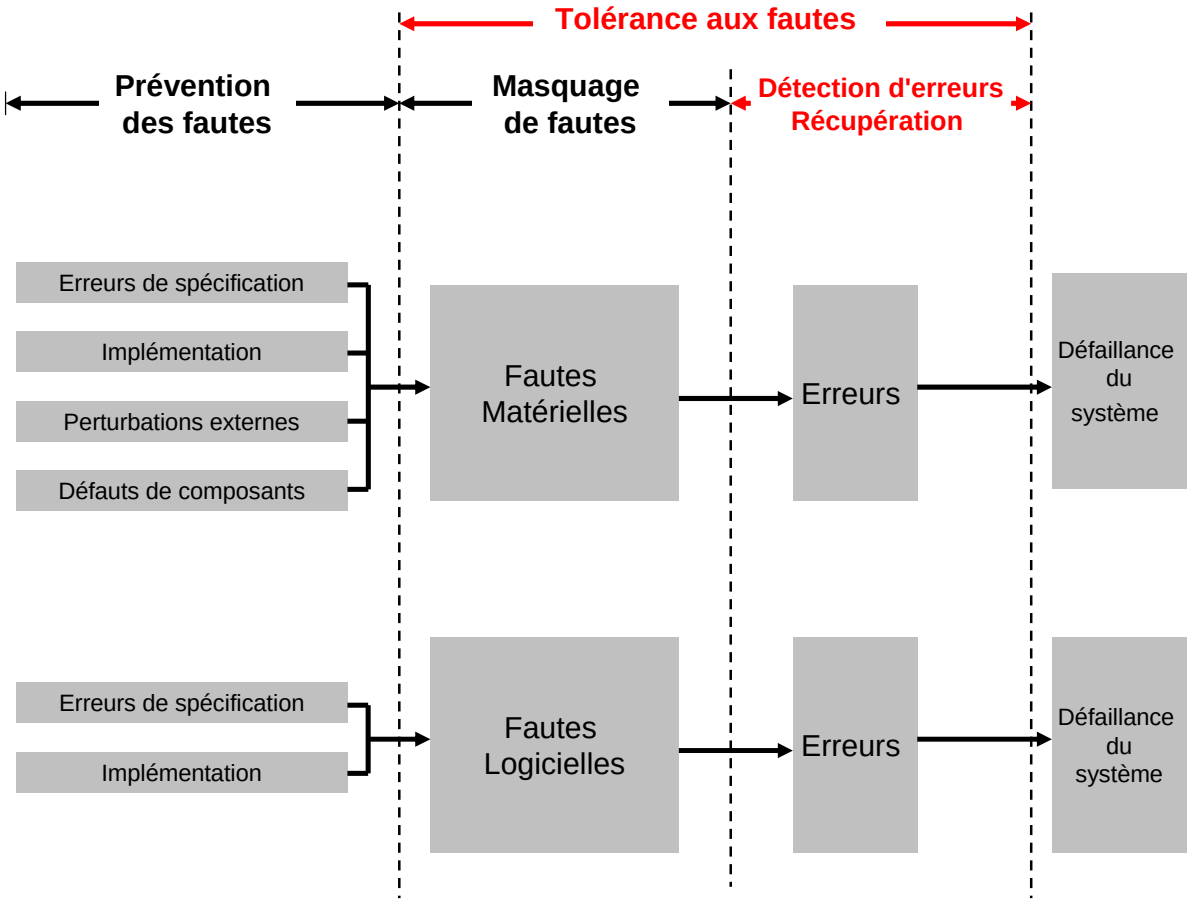


FIGURE 1.11 – Les techniques de sûreté de fonctionnement.

des capteurs électriques. Certaines techniques peuvent être ajoutées à étape de conception logique, par exemple en ajoutant de la redondance matérielle ou temporelle dans les blocs logiques et dans l'application logicielle.

La figure 1.12 est l'extension de la figure 1.2 précédemment présentée. Elle représente les différentes phases de tolérance aux fautes (détecter et corriger). Dans chaque phase, une technique différente de tolérance aux fautes peut être utilisée. Nous considérons la redondance matérielle et l'auto-vérification qui sont les deux niveaux les plus élevés (comme indiqué dans 'c' et 'd' de la figure 1.12).

1.7 Conclusion

Le but de ce chapitre a été d'introduire les concepts de sûreté de fonctionnement dans les systèmes embarqués. Dans l'accomplissement de cet objectif, nous avons introduit les principales questions liées à la conception et l'analyse des systèmes tolérants aux fautes. Nous avons présenté différents types de fautes ainsi que leurs caractéristiques, car notre objectif final est la conception d'un système de calcul tolérant aux fautes dues à des SEU.

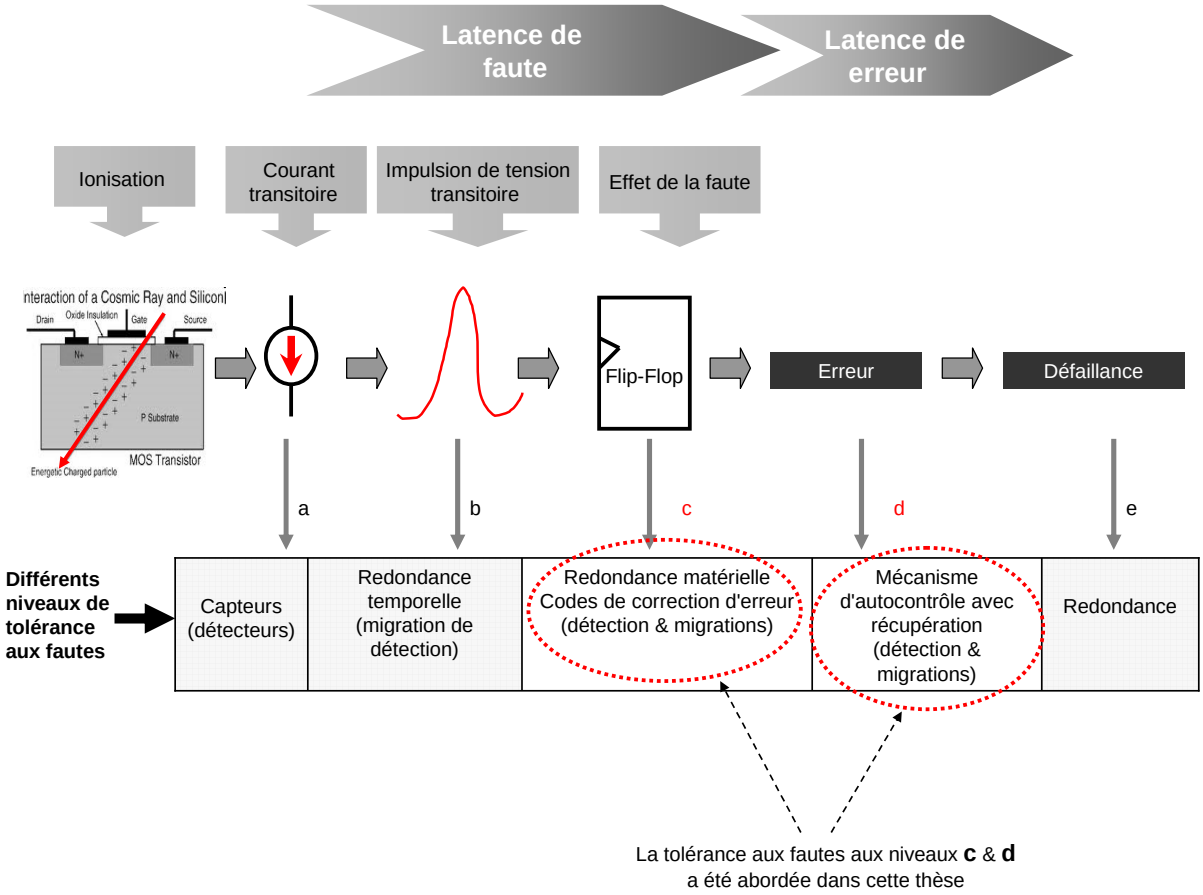


FIGURE 1.12 – Séquence des événements depuis l'ionisation jusqu'à la défaillance, et ensemble de techniques tolérantes aux fautes appliquées à différents moments. [Pie07].

En outre, ce chapitre a présenté la problématique de sûreté de fonctionnement contre les perturbations non-permanentes. Notre objectif est de proposer une méthodologie de conception de nouvelles architectures de processeurs sûrs de fonctionnement. En conséquence, dans le chapitre 2, nous allons discuter de certaines méthodologies existantes de détection et correction des erreurs.

Chapitre 2

Méthodes de conception et d'évaluation des processeurs tolérants aux fautes

Le but des techniques de tolérance aux fautes est de limiter les effets d'une faute, ce qui signifie augmenter la probabilité qu'une erreur soit acceptée ou tolérée par le système. Une caractéristique commune de toutes les techniques des tolérances aux fautes est l'utilisation de la redondance. La redondance est tout simplement l'ajout de ressources matérielles ou temporelles au-delà de ce qui est nécessaire pour un fonctionnement normal du système [Poe05]. Il peut s'agir de matériel – certains modules matériels sont dupliqués, ou tripliqués comme dans la TMR, de temps – des parties d'un programme sont exécutées plusieurs fois, de l'information – le circuit ou le programme utilise une information redondante, ou un mélange de ces trois solutions.

Les solutions traditionnelles impliquant une redondance excessive sont pénalisées en surface, en puissance et en performance [BBV+05], et d'autres approches moins coûteuses ne fournissent pas la détection des fautes et les capacités de correction nécessaires. Les systèmes embarqués tolérants aux fautes doivent être optimisés afin de répondre aux contraintes de temps et de surface [PIEP09], c'est pourquoi une attention particulière est requise lors du choix des techniques de redondance pour les applications critiques.

En conséquence, ce chapitre présente une comparaison des techniques de tolérance aux fautes existantes en termes de capacité de détection et de correction d'erreur, de performances et de surcoût matériel. De ces comparaisons, nous identifierons les techniques qui peuvent remplir efficacement nos objectifs de conception.

Par la suite, nous examinerons les techniques de redondance employées dans les différents processeurs tolérant aux fautes, et la dernière section se penchera sur les méthodes d'évaluation mises en œuvre pour vérifier l'efficacité des méthodes de conception tolérantes aux fautes dans les processeurs.

2.1 Détection d'erreurs

La détection d'erreur génère un signal d'erreur ou un message dans le système. Elle a déjà été présentée dans la section 1.5.4. Elle peut être basée sur la détection préventive ou sur une vérification concurrente. La détection préventive est essentiellement une technique hors-ligne qui a lieu alors que la prestation de service normale est suspendue, et contrôle le système pour détecter les erreurs latentes et les fautes dormantes, alors que la vérification concurrente est une technique enligne qui a lieu pendant la prestation de service normale [ALR01]. De même, Bickham définit la détection d'erreurs concurrente (*Concurrent Error Detection*, CED) comme un processus de détection et de rapport d'erreurs tout en réalisant, en même temps, les opérations normales du système [Bic10].

Les techniques de CED sont largement utilisées pour améliorer la fiabilité des systèmes [HCTS10, CTS+10, WL10]. Leurs principes de fonctionnement ont été résumés dans [MM00] où est considéré un système qui réalise une fonction (f) et produit une sortie en réponse à une séquence d'entrée. Le schéma classique de la CED contient généralement une autre unité qui – indépendamment – prédit une caractéristique particulière de la valeur de sortie du système pour chaque séquence d'entrée. Enfin, une unité de vérification compare les deux sorties pour prévoir un signal d'erreur. L'architecture globale d'un système CED est montrée dans la figure 2.1.

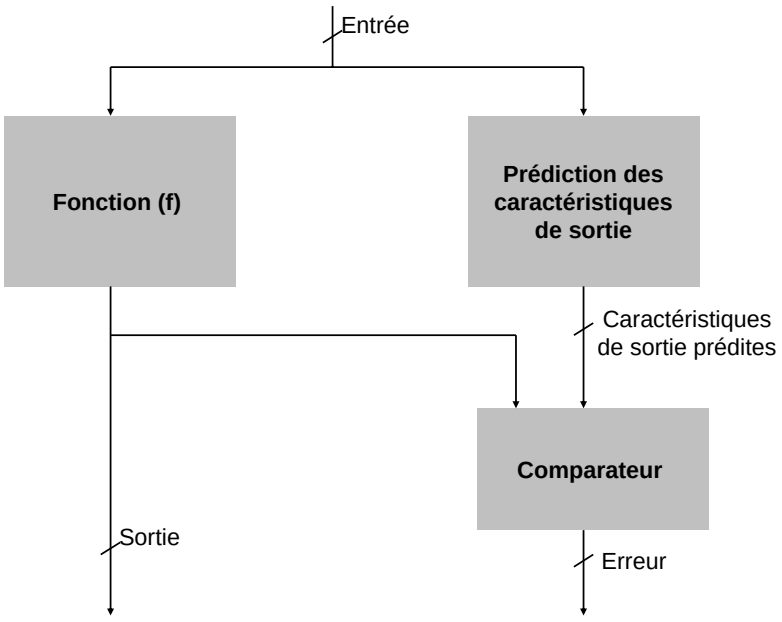


FIGURE 2.1 – Architecture générale d'un système de détection d'erreurs simultanée [MM00]

Plusieurs techniques de CED basées sur la redondance ont été proposées et utilisées commercialement pour la conception de systèmes de calcul fiables [HCTS10, SG10]. Elles ont été répertoriées en trois classes : redondance matérielle, redondance temporelle, et redondance d'informations. Un système tolérant aux fautes utilise l'une ou plusieurs d'entre elles. Ces techniques se distinguent principalement par leurs capacités de détection d'erreurs et les contraintes qu'elles imposent lors de la conception du système. Dans la section suivante, nous explorons les techniques de détection d'erreurs

couramment utilisées.

2.1.1 Redondance matérielle

La redondance matérielle est l'approche couramment utilisée [Bic10]. Elle fait référence à l'ajout de ressources matérielles supplémentaires, tel que le doublement du système, en utilisant un comparateur en sortie pour détecter les erreurs. Il est tenu compte ici de la structure du circuit et non pas de la fonctionnalité. Il est tout aussi efficace pour les fautes transitoires que pour les fautes intermittentes ou permanentes. Cependant, les exigences en surface et en puissance sont très élevées. Elle peut être classée en deux sous-types : (i) duplication avec comparaison et (ii) duplication avec redondance complémentaire.

La duplication avec comparaison (*Duplication With Comparison*, DWC) ou double redondance modulaire (*Dual Modular Redundancy*, DMR) [JHW+08] est une technique de détection d'erreur simple et facile à mettre en œuvre (voir figure 2.2). Elle possède une bonne capacité de détection d'erreur, sauf pour les bogues dus à la conception, les erreurs dans le comparateur, ou les combinaisons d'erreurs simultanées dans les deux modules. Le coût matériel du circuit augmente de plus de 100 %.

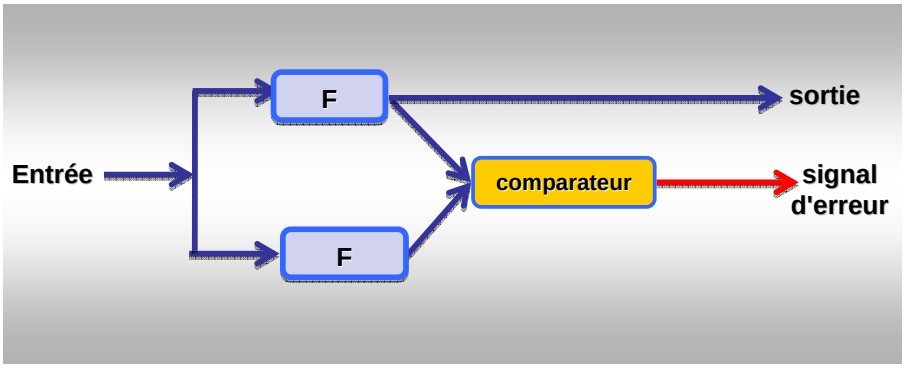


FIGURE 2.2 – Duplication avec comparaison

Il existe une autre technique complémentaire nommée « duplication avec redondance complémentaire » (*Duplication with Complement Redundancy*, DWCR) [Jab09]. Cette technique est similaire à la DWC à la différence près que les signaux d'entrée, les signaux de contrôle en sortie, ainsi que les signaux de données internes de chacun des deux modules sont de polarité opposée pour interdire les erreurs simultanées dans les deux modules afin d'éviter la défaillance du système.

Ici aussi, l'augmentation de surface et de puissance consommée est supérieure à 100 %. Cependant, cette méthode augmente la complexité de la conception par rapport à une simple duplication. Cette technique est utilisée dans le *dual-checker rail* (DCR), où les deux sorties sont inversées s'il n'ya pas d'erreur ; elle est parfois employée dans le contrôleur.

2.1.2 Redondance temporelle

Elle désigne une technique de redondance qui nécessite une seule unité effectuant une même opération deux fois de suite. Si une différence est constatée entre les deux calculs successifs, cela signifie qu'une faute transitoire ou intermittente est apparue lors de l'un ou l'autre des calculs [AFK05]. Dans cette approche, il y a une pénalité en termes de temps supplémentaire, cependant la pénalité matérielle est moindre. Il s'agit d'une technique de réplication temporelle, sans considération de la fonctionnalité du circuit.

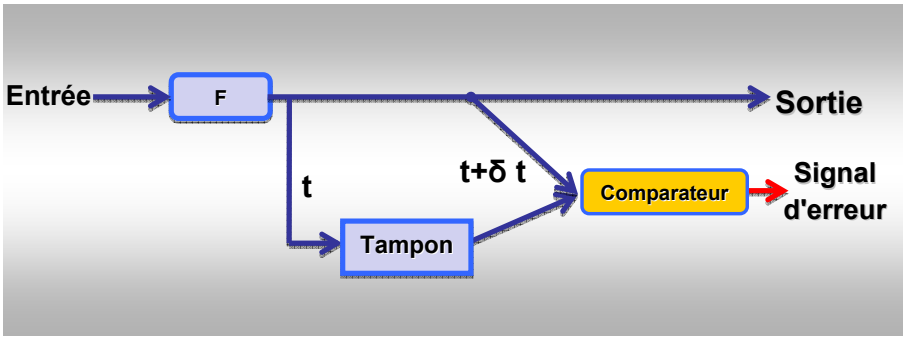


FIGURE 2.3 – Redondance temporelle pour la détection des fautes temporaires ou intermittentes

Dans cette technique, les fautes intermittentes et transitoires sont détectées (comme indiqué dans la figure 2.3) mais les fautes permanentes ne le sont pas. Pour la détection des fautes permanentes, le circuit a été modifié comme montré dans la figure 2.4 suivante, selon laquelle le calcul utilisant les données d'entrée est d'abord réalisé à l'instant t. Les résultats de ce calcul sont ensuite stockés dans une mémoire tampon. Les mêmes données sont ensuite utilisées pour répéter le calcul, en utilisant le même bloc fonctionnel à l'instant $t + \delta t$. Cependant, cette fois les données d'entrée sont d'abord encodées d'une certaine manière. Les résultats du calcul sont ensuite décodés et les résultats sont comparés aux résultats obtenus précédemment. Tout écart permet de détecter une faute permanente dans le bloc fonctionnel.

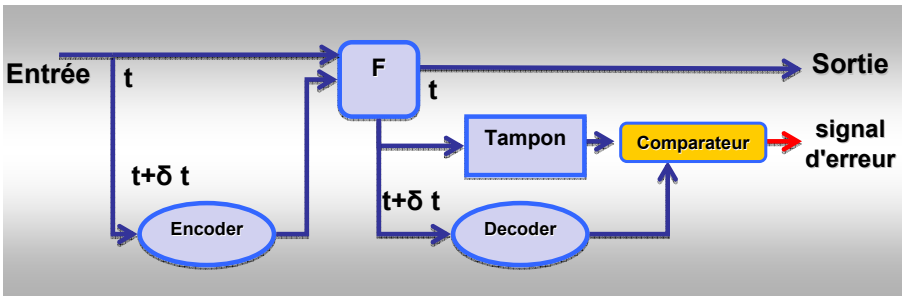


FIGURE 2.4 – Redondance temporelle pour la détection d'erreur permanente

Une approche alternative peut être l'exécution redondante avec des opérandes décalés (*Redundant Execution with Shifted Operands*, RESO) [PF06] où certaines instructions sont exécutées de manière

redondante avec des opérandes décalés sur la même unité fonctionnelles. Décaler le résultat par la même quantité produit le résultat original calculé avec les opérandes non décalés. La réexécution des instructions détecte les fautes temporaires, alors que la réexécution avec des opérandes décalés détecte également les fautes permanents. Le système fonctionne lorsque la fonctionnalité possède les propriétés requises telles que la linéarité.

La redondance temporelle affecte directement les performances du système, même si le coût matériel est généralement inférieur à celui de la redondance matérielle. Ainsi les systèmes basés sur la redondance temporelle sont comparativement plus lents. Afin de surmonter cet inconvénient, de nombreux systèmes utilisent le pipelining pour masquer le problème de latence. Les conséquences énergétiques de la redondance temporelle ne sont pas du tout traitées, à l'exception du fait qu'elle consomme deux fois plus d'énergie qu'une unité non redondante.

2.1.3 Redondance d'information

L'idée sous-jacente d'un schéma de redondance d'information est d'ajouter de l'information redondante aux données transmises, stockées ou traitées, afin de déterminer si des erreurs ont été introduites [IK03]. C'est une façon de protéger les données grâce à un codage mathématique qui peut être réutilisé ensuite pour décoder les données originales (comme montré dans la figure 2.5).

Les circuits de codage et de décodage ajoutent des délais supplémentaires, ce qui les rend plus lents que la DMR, mais le surcoût matériel est beaucoup plus faible que pour la DMR. Lors du codage, l'information stockée ou la fonctionnalité du circuit sont considérées, mais la structure du circuit n'est pas prise en compte. Typiquement, la redondance d'information est utilisée pour protéger des éléments de stockage (comme la mémoire, les caches, les piles de registres, etc.) [HCTS10] comme par exemple dans les processeurs Power 6 et 7 [KMSK09]. Ces codes sont classés en fonction de leur capacité de détection et de l'efficacité du code de correction, et de leur complexité. Dans cette section, nous discuterons seulement des codes de détection d'erreur.

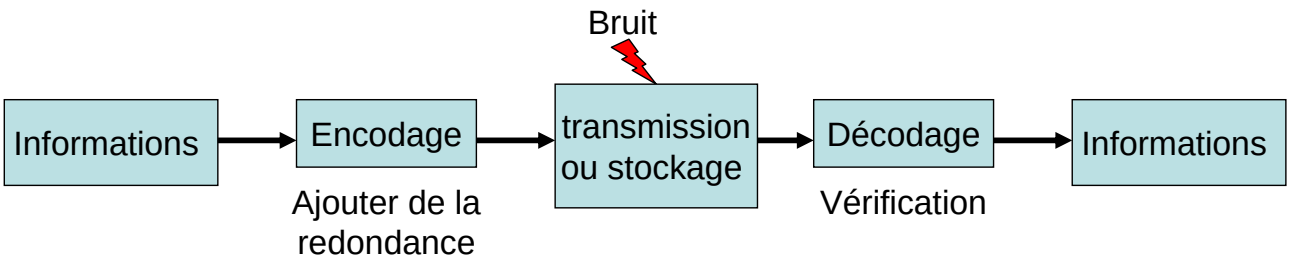


FIGURE 2.5 – Principe de la redondance d'information

Les codes de détection d'erreurs (*Error Detecting Codes*, EDC) ont un surcoût matériel moindre que les codes correcteurs d'erreurs. Il existe différents EDC, par exemple les codes de parité, de Borden, de Berger ou de Bose. Nous ne rentrerons pas dans les détails, mais nous comparerons leurs caractéristiques essentielles.

La stratégie de codage de parité est la plus simple et offre le plus faible surcoût matériel [ARM+11]. Elle est basée sur le calcul des parités paires ou impaires pour les données d'un mot de longueur N. La parité peut être calculée avec l'opération XOR entre les bits de données. Un code de parité a une distance de 2 et peut détecter toutes les erreurs impaires.

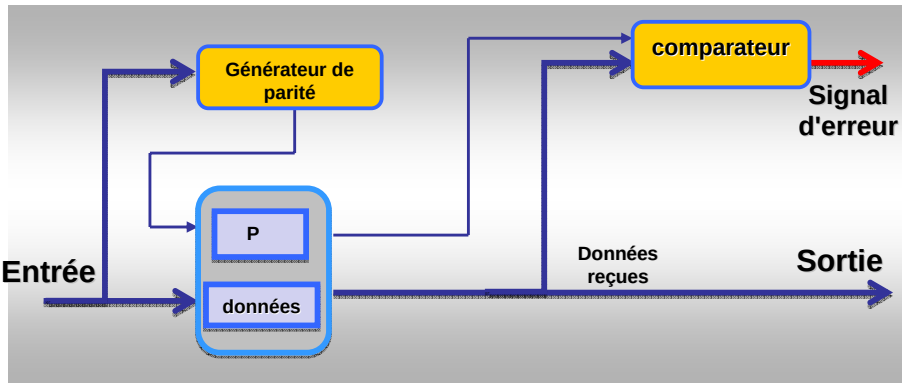


FIGURE 2.6 – Codeur de parité pour le stockage des données

Avant de stocker les données dans le registre, le générateur de parité est utilisé pour calculer le bit de parité nécessaire (comme le montre la figure 2.6). Ensuite, la parité calculée et les données originales sont stockées dans le registre. Lorsque les données sont récupérées, un comparateur de parité est utilisé pour calculer la parité à partir des bits de donnée stockés. Le comparateur de parité compare la parité calculée et la parité stockée, et un signal d'erreur est positionné en conséquence. De même, le codage de parité peut également être utilisé pour protéger les fonctions logiques (voir la figure 2.7). Il est utilisé couramment dans les processeurs pour vérifier les erreurs dans les bus, la mémoire, et les registres [IK03].

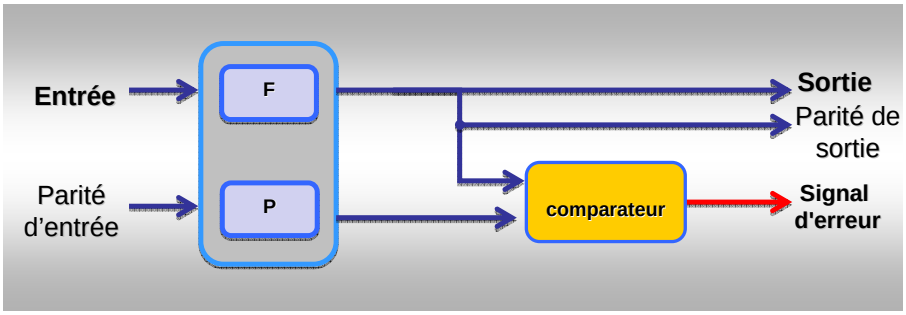


FIGURE 2.7 – Parité fonctionnelle

Les codes à redondance cyclique (*Cyclic Redundancy Check*, CRC) constituent une autre classe d'EDC. Ils sont communément employés pour détecter les erreurs dans les systèmes numériques [IK03]. Les CRC sont des codes de contrôle de parité avec la propriété supplémentaire que le décalage cyclique d'un mot code est également un mot code. Ainsi, si $(C_{n-1}, C_{n-2}, ...C_1, C_0)$ est un mot code, alors $(C_{n-2}, C_{n-3}, ...C_0, C_{n-1})$ est aussi un mot code.

L'idée est d'ajouter une somme de contrôle à la fin de la trame de données de telle manière que le polynôme représenté par la trame résultante est divisible par le polynôme générateur $G(x)$ convenu par l'expéditeur et le récepteur. Lorsque le destinataire reçoit la trame contenant la somme de contrôle, il la divise par $G(x)$ et si le reste n'est pas nul, il y a eu une erreur de transmission. Il est alors évident que les meilleurs polynômes générateurs sont ceux qui sont le moins susceptibles de diviser régulièrement en une trame qui contient des erreurs. Les CRC se distinguent par les polynômes générateurs qu'ils utilisent. Ils ne peuvent pas déterminer directement la position du bit d'erreur pendant le processus de décodage. Par conséquent, ils sont limités à la détection d'erreur uniquement.

Les codes de Borden sont une autre classe de codes qui peuvent détecter les erreurs unidirectionnelles (erreurs qui provoquent soit une transition $0 \rightarrow 1$, soit $1 \rightarrow 0$, mais pas les deux). Ce sont les codes optimaux pour la détection d'erreurs unidirectionnelles.

L'EDC de Berger est capable de détecter toutes les erreurs unidirectionnelles. Il est formulé en ajoutant un bit de contrôle au mot de donnée. Le bit de contrôle constitue la représentation binaire du nombre de 0 présents dans le mot de donnée. Par exemple, dans un mot de donnée de longueur 3 bits, nous avons besoin de 2 bits pour la vérification. Le code de Berger est plus simple à mettre en œuvre que les codes de Borden.

Le code de Bose est plus efficace que le code de Berger. Il offre la même possibilité de détection d'erreur, mais avec moins de bits de contrôle.

En résumé, en augmentant la complexité des codes, leur efficacité augmente également. Choisir le bon code dépend des besoins de l'application.

Dans les circuits de traitement arithmétique, comme dans les ALU, les codes précédents sont incapables de détecter des erreurs parce que lorsque deux symboles sont soumis à une opération arithmétique, il en résulte un nouveau symbole qui ne peut pas être exprimé uniquement par la combinaison des entrées [FP02, Nic02]. En d'autres termes, ils sont utiles dans la vérification des opérations arithmétiques, où la parité ne serait pas préservée [FP02, IK03]. Les différentes parties de l'information d'un opérande sont traitées par un opérateur arithmétique typique, tandis qu'un symbole de contrôle est généré simultanément (sur la base des bits d'information) [Bic10]. Ils ont deux implantations classiques : les codes AN et les codes à résidus.

Les codes AN sont la forme la plus simples de codes arithmétiques [Muk08]. Ils sont formés en multipliant chaque mot de données N par une constante A. L'équation suivante donne un exemple de code d'AN :

$$A\left(N_1 + N_2\right) = A\left(N_1\right) + A\left(N_2\right) \tag{2.1}$$

Ils ne sont conservés que par les opérations arithmétiques et ils ne sont pas valables pour les opérations logiques et de décalage. Ils ne sont pas couramment employés en raison de la forte pénalité tant matérielle que temporelle.

Les codes à résidus sont un autre type de code arithmétique, dans lequel l'information utilisée pour la vérification est appelée « résidu ». Le résidu r d'un opérande A est égal au reste de A divisé par le

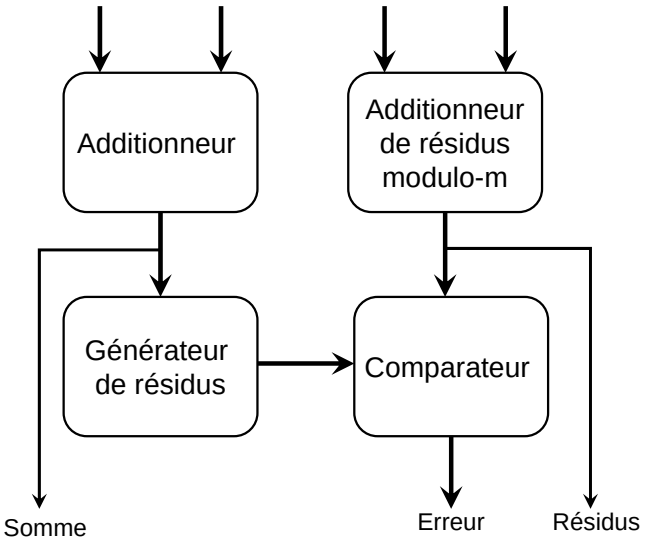


FIGURE 2.8 – Additionneur à code à résidus [FFMR09]

modulo de base m [Bic10]. Les deux calculs sont effectués simultanément (voir figure 2.8). Pour la première étape de calcul, deux opérandes A et B se voient appliquer une opération arithmétique dans l'ALU. Un générateur de résidus produit alors le code de résidus à partir du résultat d'ALU. Lors du calcul, les deux opérandes entrent simultanément dans un générateur de résidus. Ces résidus sont alors soumis à la même opération de l'ALU que pour le premier calcul (addition, dans ce cas) [FFMR09]. Enfin les résidus sont comparés afin de détecter les erreurs.

Dans ce travail, les codes de parité simple seront initialement employés pour la conception d'un cœur de processeur autocontrôlé (*Self Checking Processor Core*, SCPC). Notre principal objectif étant de prouver la validité du concept de la méthodologie proposée, un code simple comme la parité va simplifier la mise en œuvre. À un stade plus avancé, des codes alternatifs comme le CRC simplifié peuvent être utilisés pour une meilleure couverture des erreurs. À l'intérieur de l'ALU, les codes à résidus seront utilisés en plus des bits de parité, et les instructions seront regroupées en opérations arithmétiques et logiques. Un contrôle de parité simple est mis en œuvre pour la détection des erreurs dans les instructions logiques et les résidus sont utilisés pour la détection des erreurs dans les instructions arithmétiques. Ainsi, la surface du processeur sera réduite par rapport à la TMR.

2.2 Correction d'erreurs

La correction d'erreurs a déjà été discutée dans la section 1.5.4. Un système avec détection d'erreur est un système sûr, mais il ne peut continuer à fonctionner tant qu'il n'a pas la capacité de corriger les erreurs. De manière similaire à la détection des erreurs, les techniques de correction sont aussi classées en trois sous-classes : la redondance matérielle, temporelle ou d'information.

2.2.1 Redondance matérielle/statique

L'ajout d'un troisième bloc supplémentaire et le remplacement du comparateur par un voteur dans une architecture DMR produit une architecture TMR, comme indiqué dans la figure 2.9. La TMR, en plus de leur détection, peut également corriger les erreurs. Une approche plus générale de la redondance N-modulaire est discuté dans [KKB07]. Dans cette technique les effets des fautes sont masqués. Tous les composants fonctionnent simultanément et leurs sorties sont envoyées vers un voteur. La sortie du voteur sera correcte si au moins deux des composants sont non défectueux. Les techniques de redondance statique sont caractérisées pour être simples, mais elles ont un fort surcoût en surface et en consommation.

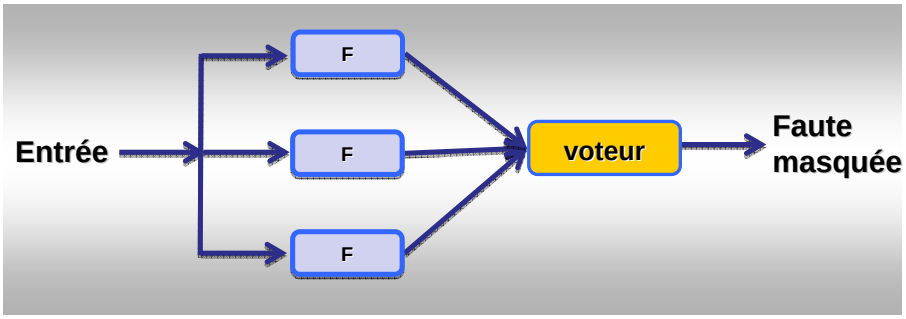


FIGURE 2.9 – Redondance modulaire triple

La technique TMR a longtemps constitué la solution de tolérance aux fautes principale dans les avions [Yeh02] et les navettes spatiales, où non seulement les processeurs, mais les systèmes entiers sont répliqués pour garantir la robustesse. La TMR peut être mise en œuvre au niveau logiciel. Une approche proposée dans [SMR+07] utilise la mise en œuvre logicielle de la TMR dans laquelle les processus du système d'exploitation sont tripliqués et exécutés sur plusieurs cœurs disponibles. La réplication d'entrée et la comparaison de sortie sont faites par une unité d'émulation d'appel système.

La TMR peut être employée pour traiter les erreurs de données à bit unique (SET, persistantes, non persistantes) survenant dans une cellule [Car01]. Dans cette technique le point faible est le voteur, parce que si une faute se produit dans le voteur le système complet échoue. Toutefois, le voteur est généralement petit, et donc souvent considéré comme fiable. Il y a une pénalité en surface et en consommation importante (un facteur 3 à 3,5 environ) associée à la TMR par rapport à la conception non-redondante [JHW+08].

2.2.2 Redondance temporelle

Pour la correction d'erreur à l'aide de la redondance temporelle, un calcul est répété sur le même matériel à trois intervalles de temps différents puis un vote intervient sur les résultats [MMPW07]. Elle exige donc trois fois plus de cycles d'horloge pour exécuter la même tâche. Elle peut uniquement corriger les erreurs dues aux fautes transitoires à condition que la durée de la faute soit inférieure au temps de calcul. Ayant besoin de temps supplémentaire pour répéter les calculs, elle ne peut être

employée que dans les systèmes avec peu ou pas de contraintes temporelles. Cependant, elle offre des surcoûts plus faibles par rapport à la TMR.

2.2.3 Redondance d'information

Les codes correcteurs d'erreurs (*Error Correcting Codes*, ECC) peuvent fournir des solutions moins coûteuses que les autres techniques de redondance connues comme la TMR [CPB+06]. Ils sont couramment utilisés pour protéger les mémoires (voir figure 2.10). Le surcoût d'un code repose sur : (i) des bits supplémentaires nécessaires pour protéger les informations ; (ii) du matériel supplémentaire et de la latence pour le codage et le décodage. Cependant, la latence du codage et du décodage peut être réduite si ceux-ci sont exécutés en parallèle.

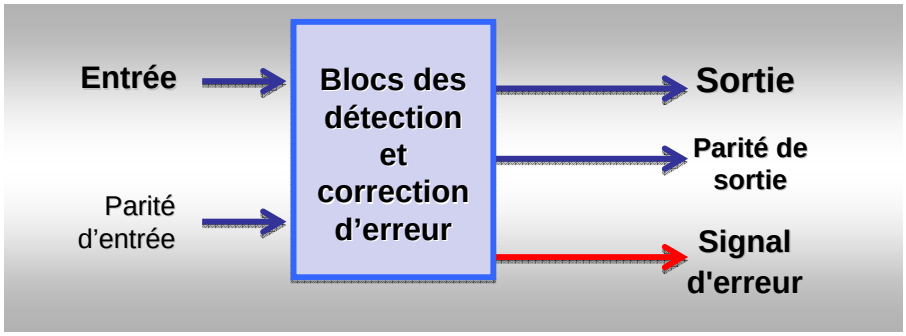


FIGURE 2.10 – Bloc mémoire à détection et correction d'erreurs

Parmi les différents ECC, les plus couramment employés dans les circuits numériques comprennent les codes de Hamming, Hsiao et Reed-Solomon. Ces codes peuvent corriger les erreurs en plus de leur détection. Il y a deux paramètres clés des codes correcteurs d'erreurs : (i) le nombre de bits erronés qui peuvent être détectés et (ii) le nombre de bits erronés qui peuvent être corrigés. Les propriétés des codes de détection et de correction d'erreurs sont basées sur leur capacité à partitionner un ensemble de $2n$ mots de N bits dans un espace de code de 2^m mots code et un espace non-code de $2^n - 2^m$ mots [FP02]. Les codes blocs les plus simples sont les codes de Hamming. Ce sont des codes capables de corriger une seule erreur, et d'en détecter deux (SEC-DED) [LBS+11], mais pas les deux simultanément. Ce sont les premiers codes ECC linéaires. Ils sont très utiles dans les cas où la probabilité d'une seule erreur est significative.

Les codes Hsiao (aussi appelés codes de Hamming évolués) sont d'autres codes communément utilisés pour la protection/correction des erreurs dans les mémoires [Mon07]. Ils permettent un encodage et une détection d'erreur plus rapide que les codes Hamming [Hsi10].

Les codes les plus puissants peuvent être construits en utilisant des polynômes générateurs appropriés. Parmi eux, les codes Reed-Solomon sont des codes cycliques qui nécessitent un circuit de codage et de décodage complexe et sont particulièrement bien adaptés aux applications où des erreurs se produisent en rafales. C'est pourquoi ils sont surtout employés dans le codage canal. D'autre part, les schémas de codage convolutifs sont utiles dans les systèmes de stockage et de transmission de

données, tels que les mémoires et les réseaux [FP02].

2.3 Recouvrement d'erreur

Le recouvrement transforme un état du système qui contient une ou plusieurs erreurs et – éventuellement – des fautes en un autre état sans erreurs détectées ni fautes qui puissent être à nouveau activées [ALR01]. Il peut seulement être initié par la détection de faute ou d'erreur, et par conséquent le système doit intégrer un mécanisme d'autocontrôle. Aujourd'hui, les microprocesseurs modernes disposent de multiples capacités de détection d'erreur intégrées, comme la détection d'erreurs dans la mémoire, le cache, les registres, détection d'opcodes illégaux, etc. [MBS07]. Le recouvrement peut être basé sur la gestion des erreurs à haut niveau (éliminer les erreurs des états du système) ou sur la gestion des fautes bas niveau (empêcher la faute d'être à nouveau activée).

Le recouvrement masque les conséquences des fautes à l'utilisateur. Il est plus adapté aux fautes temporaires et intermittentes que pour les fautes permanentes, pour lesquelles le recouvrement n'est généralement pas suffisant. Il a besoin d'une propriété obligatoire qui est la gestion de fautes (voir la figure 2.11), il élimine les fautes de l'état du système [LB07]. La fonction de gestion des fautes empêche les fautes d'être à nouveau activées. Cela nécessite d'autres caractéristiques telles que le diagnostic, qui révèle et localise la ou les cause(s) d'erreur(s) [LRL04]. Le diagnostic peut également activer un traitement des erreurs plus efficace. Si la cause de l'erreur est localisée, la procédure de recouvrement peut prendre des mesures concernant les composants associés, sans affecter les autres parties du système et les fonctionnalités liées au système. Dans ce travail, nous nous pencherons sur les erreurs temporaires dues à des fautes transitoires. Par conséquent, il ne sera pas important d'explorer les techniques de gestion de fautes.

Il existe deux sous-types de recouvrement des erreurs ; le recouvrement d'erreur direct (*Forward Error Recovery*, FER) et recouvrement d'erreur arrière (*Backward error Recovery*, BER). En FER, le système n'a pas besoin de restaurer son état et continue à progresser sans restaurer les états du système. Le « compensateur » permettra de surmonter les fautes, comme le montre la figure 2.11 (FER). Par exemple, dans la technique TMR le voteur masquera (compensera) la faute, et dans les techniques à base d'ECC, le circuit de correction d'erreur corrigera l'erreur, si elle est corrigible.

Le BER implique la restauration de l'état du système vers un état précédent connu sûr. Pour que le BER soit un succès, le système doit être au courant des faits suivants : (i) lors d'un point de recouvrement quels états doivent être sauvegardés et où ; (ii) quel algorithme utiliser ; et (iii) que doit faire le système après le recouvrement. De même, pour les fautes transitoires le BER sera suffisant, alors que la gestion de faute est une caractéristique nécessaire pour les fautes permanentes (voir la figure 2.11).

Il y a deux algorithmes connus pour sauver les états de recouvrement du BER : le « *checking point* » et le « *logging* ». Le choix dépend de la « microarchitecture » du cœur et des exigences de recouvrement, car les deux ont des coûts différents pour différents types d'états, et de nombreux systèmes BER utilisent une méthode hybride. Un système présenté dans [SMHW02] utilise un BER

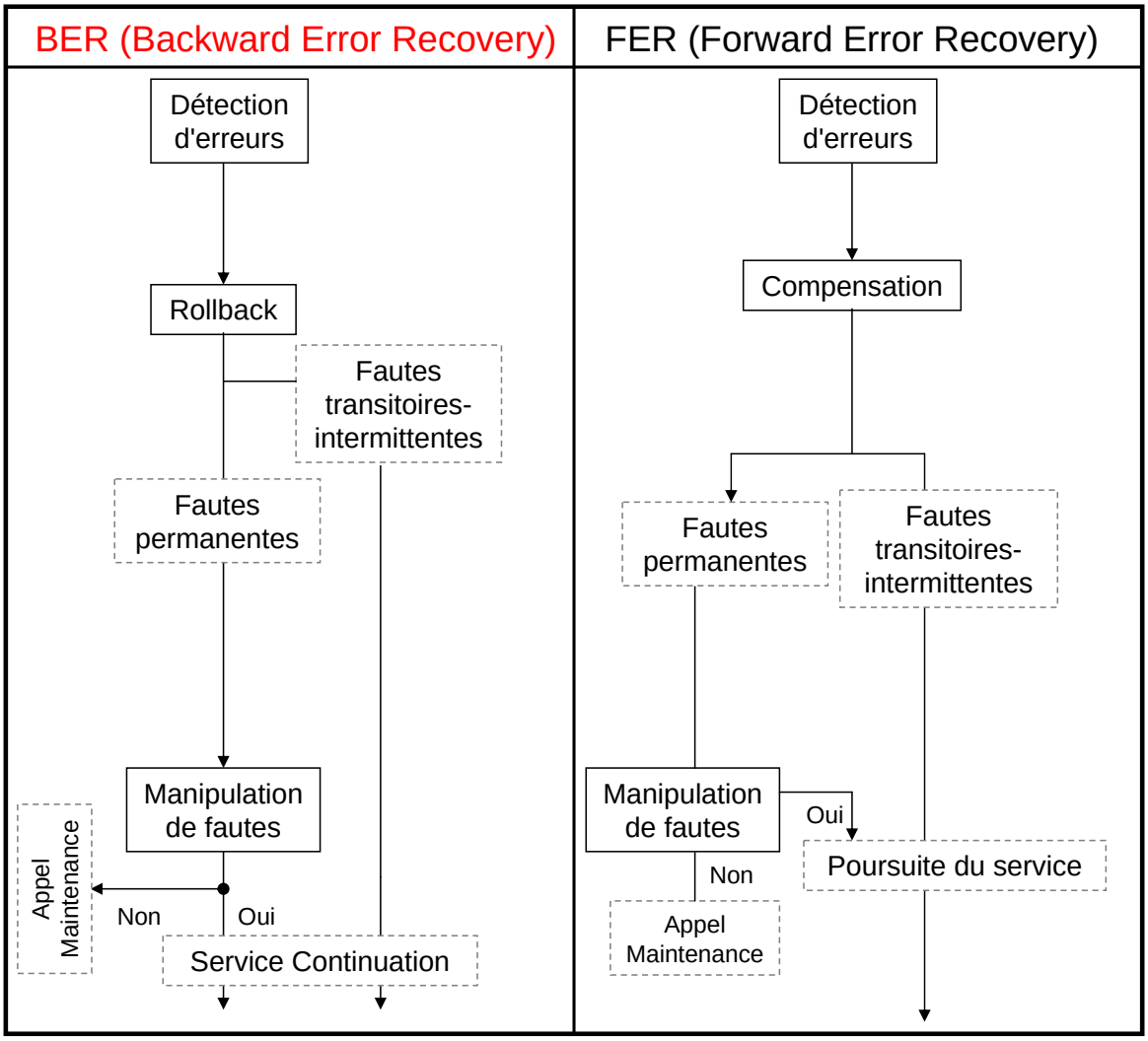


FIGURE 2.11 – Stratégies de base pour la mise en œuvre du recouvrement d'erreur.

hybride. Un réel critère de choix est que si nous avons peu de registres et que les recouvrements ne sont pas fréquents, alors le « *checking point* » sera préféré. S'il y a beaucoup de registres et que les recouvrements sont fréquents, alors on privilégiera le « *logging* ».

Un autre aspect important est l'endroit où sauvegarder les états lors du point de recouvrement. Une copie (*shadow*) de la file de registres est créée dans le cœur pour sauvegarder les états internes. Les valeurs sauvegardées dans la copie peuvent être utilisées pour le rollback et le recouvrement [AHHW08]. Toutefois, d'autres techniques, qui exigent une haute fiabilité, stockent l'état des registres internes à l'extérieur de la puce. Lorsque les états sont récupérés, des ECC sont utilisés pour éviter de possibles erreurs. Dans la période récente, de nombreux développements ont été faits en BER, et de nombreux processeurs à faible coût emploient cette technique, comme par exemple IBM qui a intégré le recouvrement dans la microarchitecture POWER-6 [MSSM10].

2.4 Tendances de conception des processeurs tolérants aux fautes

Depuis peu, les architectures de calcul tolérantes aux fautes ont commencé à attirer l'attention d'un éventail de plus en plus large de communautés industrielles et académiques, en raison d'une demande accrue en sécurité et en fiabilité [ZJ08]. Aujourd'hui, la tolérance aux fautes est un besoin des applications industrielles temps-réel [RI08]. La plupart du temps, des solutions coûteuses ne sont pas acceptables pour l'industrie, et en conséquence les processeurs modernes évitent les solutions de réplication matérielle et ont tendance à employer des techniques alternatives à moindre surcoût de consommation et de surface, comme la redondance d'information ou la redondance hybride. La redondance d'information (comme l'emploi de codes ECC) a un moindre surcoût matériel, mais elle peut entraîner une pénalité de performance supplémentaire.

La pénalité de performance et le surcoût en surface dépendent du type de code ECC. Le choix de l'ECC dépend de trois contraintes : la consommation, la surface et le taux de couverture d'erreurs. Les codes ayant le meilleur taux de couverture ont souvent une pénalité temporelle et un surcoût matériel plus élevés. Les codes de parité sont plus rapides et ont faible coût matériel alors que les codes ECC couramment employés, comme les codes de Hamming, ont un meilleur taux de couverture d'erreurs (comme le DED-SEC).

Les impacts sur la performance peuvent être minimisés dans une certaine mesure en calculant les bits de parité en parallèle. D'autre part, une tendance commune pour réduire la pénalité matériellle est de faire un compromis sur le taux de couverture et employant des codes détecteurs d'erreur à faible coût (par exemple, simple parité ou codage modulo 3). De même, certains processeurs bien connus de la dernière décennie, comme le Power 6, la série Itanium, et les SPARC 64 V emploient des prédicteurs de parité et des codes modulo dans leurs unités arithmétiques et logiques pour réduire le coût en consommation et en surface. Les codes ECC sont couramment utilisés pour protéger les caches et le stockage de données [QLZ05]. Par exemple, le processeur Itanium peut détecter des erreurs de 2 bits dans le cache en s'appuyant sur les ECC. Les processeurs IBM ont des caches L1 à écriture directe (*write through*) qui utilisent la parité simple, alors qu'un ECC est utilisé dans la mémoire cache L2. D'un autre côté, Intel utilise des codes ECC même dans les caches L1.

L'utilisation de points de contrôle et du rollback est une tendance alternative. Elle peut être une solution efficace pour les processeurs tolérants aux fautes qui ont peu d'états internes (ou registres). Plus le nombre d'états internes est élevé, plus grande sera la pénalité en performance (ou en temps) pour la vérification, le chargement et la sauvegarde des états. Certains processeurs modernes qui emploient cette méthode réduisent la pénalité en définissant un check-point après chaque bloc d'instruction super-scalaire, comme par exemple les processeurs Power 6 et Power 7.

Une nouvelle tendance pour la conception d'un processeur est d'employer une couverture d'erreurs souple qui permet à l'utilisateur de choisir le niveau de protection et de redondance nécessaires pour une application particulière, comme par exemple la série ARM Cortex-R (un processeur d'application spécifique). Pour une couverture plus élevée, la DMR est employée alors que pour une couverture plus faible les ECC sont utilisés. Toutefois, le surcoût en surface sera toujours supérieur à

200 %. Dans la section suivante, nous discutons des différentes méthodologies de tolérance aux fautes mises en œuvre dans certains processeurs bien connus de la dernière décennie.

SPARC 64 V [AKT+08]

Le microprocesseur SPARC 64 V est destiné aux serveurs UNIX à mission critique. Afin de permettre un fonctionnement ininterrompu, ces serveurs doivent être résistants aux erreurs *soft*. En outre, l'intégrité des données est très importante en raison des dangers que la corruption silencieuse de données (*Silent Data Corruption*, SDC) peut poser dans les systèmes critiques. Pour répondre à ces exigences, ce processeur a été conçu non seulement pour corriger les erreurs en SRAM, mais aussi pour détecter les erreurs dans les circuits logiques, ainsi que pour recouvrer ces erreurs lorsque c'est possible.

Il dispose de trois petites mémoires caches de 128 Ko chacune, à savoir le cache d'instructions niveau 1 (L1), le cache de données niveau 1 et l'historique des branchements (*BRanch HIStory*, BRHIS). Le cache de données L1 est à écriture différée (*write back*) et protégé par les mêmes codes SEC-DED que le cache de niveau 2. Le cache d'instructions L1 et le BRHIS sont couverts par un contrôle de parité. Quand une erreur est détectée pendant la lecture du cache d'instructions L1, la donnée lue est invalidée et rappatriée du cache L2 protégé par ECC. Une erreur dans BRHIS est traitée comme un défaut de cache, et le processeur retarde l'exécution de l'instruction de branchement conditionnel jusqu'à ce que l'adresse de branchement correcte soit calculée. Le processeur perd légèrement en performances, mais est capable de continuer l'exécution des instructions correctement.

Les circuits logiques du processeur sont protégés par des octets de contrôle de parité pour détecter les erreurs logiques de 1 bit dans chaque octet. Les bits de contrôle de parité sont calculés en même temps que la génération des nouvelles données et transmis avec les données associées à travers les circuits logiques du processeur. Les bits de parité sont contrôlés à la réception.

Les unités arithmétiques et logiques sont équipées de prédicteurs de parité. Ces prédicteurs de parité calculent les bits de parité pour chaque octet de sortie de l'unité arithmétique et logique en utilisant les mêmes signaux d'entrée que l'unité à contrôler. Ces bits de parité calculés indépendamment sont comparés avec les bits de parité calculés à partir de la sortie de l'unité arithmétique et logique. Les multiplieurs sont vérifiés avec un système modulo 3.

Les prédicteurs de parité dans l'unité arithmétique et logique ne détectent pas les erreurs ponctuelles qui résultent en l'inversion d'un nombre pair de bits dans l'octet de sortie, et le modulo 3 utilisé dans les multiplieurs ne détecte pas les erreurs ponctuelles qui produisent le même résidu modulo 3. Ces contrôles, cependant, détectent la majorité des erreurs et sont économiques par rapport à une implantation de type duplication complète et comparaison. Quand une erreur de parité est détectée dans les circuits logiques ou les petits réseaux de cellules SRAM, le processeur stoppe l'émission de nouvelles instructions et efface tous les états intermédiaires. Il reprend ensuite l'exécution à l'instruction immédiatement successive à la dernière instruction exécutée correctement en utilisant les états vérifiés lors du checkpoint. Cette action est appelée retentative.

Les mécanismes de checkpoint et de retentative sont mis en œuvre dans le processeur pour le recouvrement lors d'erreurs de prédiction de branchement. Ainsi, le coût additionnel lié à l'utilisation de ces mécanismes pour le recouvrement d'erreur est faible. Par ailleurs, de nombreux microprocesseurs disposent aujourd'hui soit d'ECC soit d'octettes de parité pour les grands réseaux de cellules SRAM sur puce. Par rapport à ces microprocesseurs, le SPARC 64 V ne nécessite que quelques transistors supplémentaires pour mettre en œuvre les bits de parité, les prédicteurs de parité et les contrôleurs de parité associés dans les circuits logiques et les petits réseaux SRAM. Le nombre de transistors consacrés aux mécanismes de détection d'erreur du microprocesseur SPARC 64 V représent environ 10 % des transistors pour les portes logiques, les bascules à verrouillage, et les petits réseaux SRAM protégés par parité.

LEON 3 FT

Le LEON 3 est le successeur du processeur LEON 2 développé pour l'agence spatiale européenne (*European Space Agency*, ESA). Le LEON 3 FT [GC06] est une version tolérante aux fautes du LEON 3 standard, clone du SPARC V8. Dans le LEON 3 FT, la considération est uniquement portée sur la protection du stockage des données et non sur la fonctionnalité du processeur. Il n'existe aucune protection pour l'unité de contrôle, le chemin de données et les circuits d'ALU.

Les registres internes sont protégés par des codes ECC, ainsi que des registres de copie. Dès qu'une erreur de parité est détectée, un duplicata de la donnée est lu à partir d'une zone redondante de la file de registres, remplaçant les données dont la lecture a échoué. Quelques registres internes ont une capacité de détection d'erreur de quatre bits, la majorité des registres ayant cependant une capacité de détection d'erreur de deux bits uniquement.

La mémoire cache du LEON 3 FT est constituée de caches données et instructions séparés de 8 kO chacun. Chaque cache est composé de deux parties : tags et données. Les mémoires des tags et des données sont mises en œuvre grâce à des blocs de RAM intégrés sur la même puce, et protégées par quatre bits de parité par mot de 32 bits, permettant la détection de quatre erreurs simultanées par mot du cache. Quand une erreur est détectée, la ligne de cache correspondante est effacée et l'instruction est relancée. Cette opération nécessite 6 cycles d'horloge (états inactifs) et est transparente pour le logiciel. À des fins de diagnostic, des compteurs d'erreurs sont prévus pour surveiller les erreurs détectées et corrigées dans les régions du cache dédiées aux tags comme aux données.

Le système de contrôle du Boeing 777

Dans le Boeing 777, le système de contrôle est rendu fiable grâce à des canaux redondants utilisants des processeurs différents et des logiciels différents pour se protéger contre les erreurs de conception ainsi que les défauts du matériel [BT02]. Il utilise une triple redondance modulaire triple (triple TMR) hétérogène [Yeh02], comme le montre la figure 2.12). Trois architectures de processeurs différentes (Intel 80486, Motorola 68040 et AMD 29050) exécutent la même opération. Cependant,

c'est une solution coûteuse qui ne peut être employée que dans les applications à mission critique.

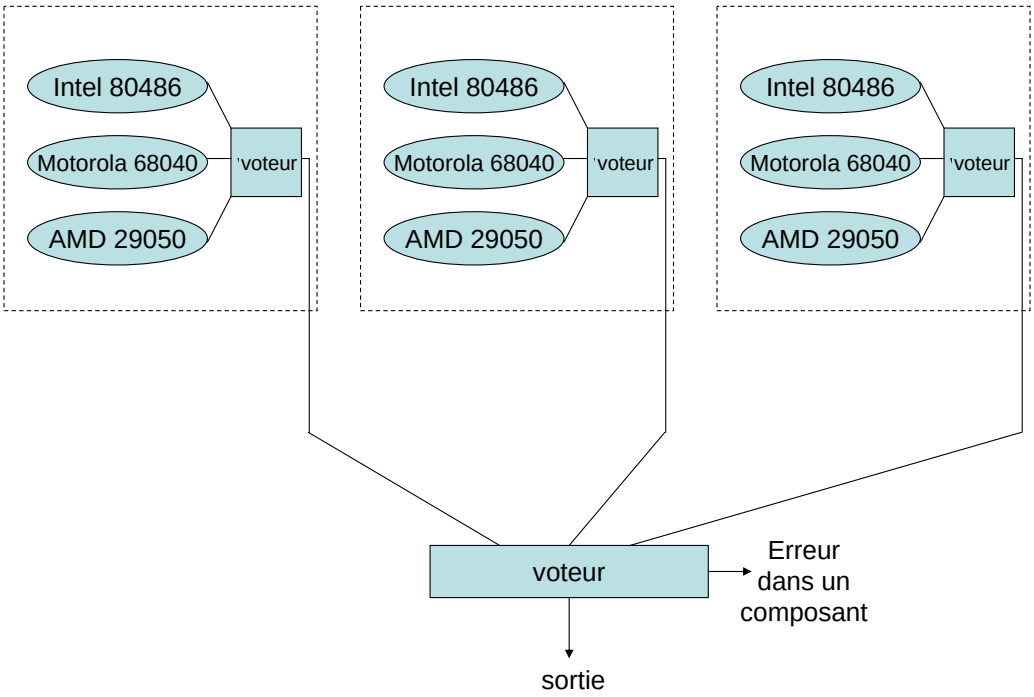


FIGURE 2.12 – La triple TMR dans le Boeing 777 [Yeh02]

ARM Cortex R Series [ARM09]

La série ARM Cortex R est une famille de processeurs embarqués pour les applications temps-réel industrielles. Ils ont un degré de personnalisation élevé, de telle sorte que le fabricant peut choisir les caractéristiques qui répondent aux besoins de ses applications.

Si l'option de fabrication « ECC » est activée, alors un système ECC de 64 bits protège le cache d'instructions : la RAM de données inclut 8 bits de codes ECC pour 64 bits de données. Le cache de données est protégé par un système ECC de 32 bits : la RAM de données inclut 7 bits de code ECC pour 32 bits de données.

Si l'option de fabrication « parité » est activée, alors le cache est protégé par des bits de parité. Pour les deux caches d'instruction et de données, les données comprennent un bit de parité par octet de données.

Le processeur peut être implanté avec une copie redondante de la plupart de la logique. Le deuxième exemplaire du cœur partage la mémoire cache avec le cœur principal – ou cœur maître, de sorte qu'un seul ensemble de cache est utilisé. La comparaison des sorties du cœur redondant avec celles du cœur principal détecte les fautes.

IBM Power 6 [MSSM10, KMSK09, KKS+07]

IBM a conçu le processeur Power 6. Au lieu de la technique TMR, il utilise des contrôleurs enligne qui consomment moins d'énergie et nécessitent moins de matériel. Il possède une capacité d'autocontrôle intégrée dans les chemins de données et de flot de contrôle. Le test des résidus est employé pour l'unité à virgule flottante et des contrôleurs de cohérence logique le sont pour la logique de contrôle. Il dispose d'une unité de recouvrement dont les points de contrôle sont effectués après un groupe d'instructions super-scalaires. Les contrôleurs enligne écrivent dans un registre d'isolement des fautes qui décide si l'état actuel ne comporte aucune erreur. En cas de détection d'erreur, l'unité de recouvrement initie une retentative d'exécution des instructions. Le bus mémoire, y compris les unités d'entrée-sortie, est protégé par des codes ECC. Le cache L1 est protégé par une parité simple, tandis que les caches L2 et L3 et tous les signaux à l'intérieur ou à l'extérieur de la puce vers le cache L3 ont une protection par ECC.

Intel Itanium 9300 [Int09]

Les processeurs de la série Intel Itanium 9300 sont des processeurs de haute performance. Les caches L2, L3 et l'annuaire du cache sont protégés par un code ECC qui peut corriger toutes les erreurs uniques et la plupart des erreurs doubles. Par ailleurs, le support du *scrubbing*[1] matériellement assisté est disponible pour les caches L2,L3, ainsi que l'annuaire. La mémoire est également protégée contre les problèmes thermiques. Ici, différents capteurs thermiques envoyant des informations aux contrôleurs mémoire qui augmentent la vitesse du ventilateur en conséquence pour réguler la température. Les registres internes du processeur sont protégés par ECC. En outre il dispose d'une horloge redondante, et des bascules et des registres durcis offrent une résistance accrue aux erreurs *soft*.

2.5 Évaluation de la tolérance aux fautes

Dans l'industrie des semi-conducteurs, les frais de test augmentent le coût de la conception de circuits intégrés et de leur fabrication. Généralement, les essais industriels sont destinés à trouver les fautes permanentes qui peuvent être produites au moment de la fabrication. Cependant, les fautes les plus fréquemment rencontrées dans les systèmes informatiques sont des effets temporaires comme les fautes transitoires ou intermittentes. Ils sont la principale cause de défaillance des systèmes numériques [VFM06]. En raison de l'augmentation de la probabilité des fautes transitoires dans les toutes dernières technologies, les concepteurs doivent de plus en plus analyser l'impact potentiel de ces fautes sur le comportement des circuits.

Le modèle d'erreur utilisé pour évaluer les fautes dépend de leur durée. Les fautes permanentes peuvent être tolérées en remplaçant le composant défectueux alors qu'une faute transitoire peut s'auto-réparer. Les fautes intermittentes sont traitées comme des fautes permanentes ou transitoires en fonction de leur fréquence. Certaines techniques courantes d'évaluation des systèmes tolérants

[1] *scrubbing* : balayage, nettoyage régulier du cache par détection et correction à la volée des erreurs

aux fautes sont traitées dans [WCS08]. Entre autres techniques, l'injection de fautes est largement acceptée comme une approche efficace pour évaluer la tolérance aux fautes [LN09, Nic10].

2.5.1 Injection de fautes

L'injection de fautes est une technique de validation pour les systèmes tolérants aux fautes qui consiste en la réalisation d'expériences contrôlées où les observations du comportement du système en présence de fautes sont induites explicitement par l'introduction volontaire de fautes dans le système [ACC+93]. En d'autres termes, c'est l'introduction délibérée de fautes (ou erreurs) dans une cible [NBV+09]. Ainsi, c'est une activation volontaire de fautes dans le but d'observer le comportement du système en leur présence. L'objectif est de comparer le comportement nominal du circuit (sans injection de faute), avec son comportement en présence de fautes injectées pendant l'exécution d'une application.

Les techniques d'injection de fautes sont devenues populaires pour évaluer et améliorer la fiabilité des systèmes embarqués à base de processeur [LAT07]. Elles peuvent être réalisées au niveau physique, ou simulées.

1. *Injection de fautes physique :* les fautes sont directement injectées dans le matériel et perturbent l'environnement (comme le bombardement d'ions lourds, les interférences électromagnétiques, le laser, etc.) [BGB+08, Too11]. De nombreuses méthodes ont été proposées, basées principalement sur la validation des systèmes physiques, incluant les injections sur les broches des circuits, l'injection d'ions lourds, la perturbation de l'alimentation, ou le laser [GPLL09]. Aucune de ces approches ne peut être utilisée pour l'évaluation de la sécurité avant que le circuit ne soit fabriqué. Par conséquent, la solution alternative est d'employer des techniques d'injection qui permettent une analyse à une étape antérieure de la conception, généralement au niveau transfert de registre ou au niveau porte ; par exemple on peut inclure des erreurs dans une description RTL.

2. *Injection de fautes simulée :* les campagnes d'injection de fautes peuvent être effectuées en utilisant plusieurs méthodes, en particulier les approches de simulation haut niveau. Cette méthode a été largement utilisée pour sa simplicité, sa polyvalence, et sa contrôlabilité [NL11]. La simulation est plus coûteuse en temps, cependant elle peut permettre une analyse plus complète et fournir des résultats plus précis et revenir moins cher que l'injection de fautes physique [NL11]. L'accès précis aux états internes du processeur est facilement réalisable par injection de fautes simulée, et c'est pourquoi elle offre une meilleure contrôlabilité/observabilité. Dans cette technique, le système sous test est simulé dans un système informatique, et les fautes sont produites en modifiant les valeurs logiques lors de la simulation.

L'injection de fautes simulée est un cas particulier de l'injection d'erreurs *soft* qui autorise différents niveaux d'abstraction du système tels que logique, architectural ou fonctionnel [CP02], et pour cette raison elle est largement utilisée. Par ailleurs, il existe plusieurs autres avantages à cette

technique. Par exemple, son plus grand avantage sur les autres techniques est la contrôlabilité et l'observabilité de tous les composants modélisés. Un autre de ses aspects positifs est la possibilité de poursuivre la validation du système pendant la phase de conception avant d'aboutir à une architecture finalisée. D'autres approches d'environnements physiques ou simulés pour effectuer des analyses de sécurité ont été discutées dans [Bau05, RNS+05].

2.5.2 Modèle d'erreur

Pour concevoir un système tolérant aux fautes, il est important que le système soit conscient des failles susceptibles d'apparaître. Certaines des fautes survenant fréquemment sont indiquées dans le tableau 2.1. Cependant, l'architecture est normalement conçue pour surmonter les erreurs potentielles. De tels systèmes peuvent détecter les fautes actives qui produisent des erreurs parce qu'ils ne sont pas conscients des phénomènes physiques sous-jacents.

TABLE 2.1 – fault modeling

Niveau	Modèle
Programmation	instruction, séquence, etc.
HDL	fonctionnel, registre
Logique	porte
Electronic	transistor CMOS
Technologie	physique, plan de masque

Il existe différents types de modèles d'erreurs et ils ont été répertoriés en trois axes dans [Sor09] : type d'erreur, durée d'erreur, et nombre d'erreurs simultanées.

Un modèle d'erreur communément considéré est le modèle de liaison qui considère les courts-circuits et la diaphonie. Ce modèle est adapté pour détecter les fautes de fabrication qui peuvent causer un court-circuit entre deux connexions. Il s'agit d'un modèle d'erreur de bas niveau.

Le modèle d'erreur *fail-stop* est un modèle d'erreur de niveau supérieur. Tous les composants vont cesser de fonctionner en cas de détection d'erreur dans un système basé sur ce modèle. Ces systèmes sont utilisés dans les systèmes critiques tels que les automates bancaires, où une seule erreur de calcul peut entraîner des centaines de dollars de perte. Un tel système s'arrête de fonctionner si des erreurs non corrigibles sont détectées.

Le modèle d'erreur de retard est celui dans lequel le circuit produit une réponse correcte, mais après un certain retard imprévu. Ce type d'erreur peut se produire en raison de divers phénomènes physiques internes au composant. Certains travaux connexes sont discutés dans [EKD+05].

Ici, nous nous sommes intéressés à des erreurs d'inversion de bits qui sont largement représentatives des erreurs temporaires dues aux SEU (SBU et MBU). Par ailleurs, elles sont faciles à modéliser à de nombreux niveaux d'abstraction.

2.5.3 L'environnement d'injection de fautes

Un environnement d'injection de fautes a généralement besoin d'au moins trois types d'informations :

(a) *quand :* quand l'injection de fautes sera-t-elle faite ? Quel est l'état qui va déclencher l'injection de fautes pendant la simulation ?

(b) *où :* où la faute sera-t-elle injectée ? Dans quelle partie du circuit ?

(c) *quoi :* quel sera le type de faute injectée ? Quels seront ses effets ?

Déclenchement de faute (quand)

Lorsque l'on étudie le comportement d'une application en présence de fautes, on peut vouloir faire l'injection de fautes de manière déterministe ou non-déterministe. Le déclenchement de faute non-déterministe peut injecter une faute lors d'une simulation, après un certain laps de temps. Dans ce cas, nous ne savons pas dans quelle partie exacte de l'application la faute est injectée. L'approche non-déterministe du déclenchement de faute consiste à compter le nombre d'instructions simulées. La faute sera injectée après un nombre prédéterminé d'instructions. Dans l'injection de fautes simulée, le comportement non-déterministe est obtenu en définissant aléatoirement le retard ou le nombre d'instructions simulées.

Un déclenchement de faute déterministe permet de limiter la portée de l'injection de fautes, en précisant que l'injection ne sera effectuée que dans un intervalle spécifique, par exemple uniquement si le pointeur d'instruction du processeur est dans une gamme de valeurs bien définie. En pratique, la solution consiste à utiliser un approche non-déterministe en la combinant avec des conditions de déclenchement, ce qui permet de gagner du temps dans certaines situations très spécifiques, en évitant que des injections non désirées ne se déclenchent pendant les campagnes d'injection de fautes.

Localisation de faute (où)

Dans notre travail, nous avons affaire à l'injection de fautes dans un processeur tolérant aux fautes. Une faute se produisant dans un processeur peut affecter l'ALU, les registres internes, ou les adresses mémoires, selon le résultat de l'instruction utilisant la logique affectée. Dans tous les cas, le changement dans les registres du processeur ou de mémoire peut représenter une faute réelle possible. La localisation d'une faute est souvent décrite de façon déterministe, mais elle peut également être décrite de manière non-déterministe si nous laissons l'environnement d'injection de fautes choisir au hasard dans quel registre du processeur la faute sera injectée.

Effet de la faute (quoi)

Comme expliqué précédemment, l'effet le plus fréquent d'une faute transitoire dans un registre du processeur ou dans la mémoire est une inversion de l'état d'un bit. En inversant un bit dans un

registre ou dans une adresse mémoire, nous pouvons injecter une faute comme si elle se produisait en situation réelle. La valeur du bit altéré est toujours basculée à la valeur opposée. Ce modèle de perturbation (*upset model*) est le modèle standard de faute transitoire utilisé dans la littérature sur la fiabilité [Muk08]. Une faute déterministe peut être générée en spécifiant quel bit inverser, mais elle peut également être faite de manière non déterministe en laissant l'environnement d'injection de fautes choisir au hasard le bit à inverser.

Les détails mentionnés ci-dessus sont les informations de base nécessaires pour le simulateur d'injection de fautes. Nos choix exacts seront présentés dans le chapitre 6, où la méthodologie de validation exacte basée sur l'injection d'erreur artificielle sera explorée.

2.6 Conclusion

Le but de ce chapitre était d'étudier les méthodologies de conception existantes et les techniques de validation d'un processeur tolérant aux fautes. Aujourd'hui, les processeurs tolérants aux fautes mettent en œuvre diverses techniques de redondance et présentent chacun leur propre surcoût en surface ou en temps. Normalement, la redondance matérielle offre une détection et une correction d'erreurs plus rapides, mais souffre d'un surcoût élevé en surface, alors que les techniques de redondance temporelle génèrent des surcoûts en surface inférieurs, mais sont associées à des surcoût temporels élevés.

Dans le passé, les processeurs tolérants aux fautes étaient uniquement utilisés pour des applications critiques et reposaient essentiellement sur des techniques de réplication du matériel. Aujourd'hui, la tolérance aux fautes est un besoin pour toute application industrielle temps-réel. Des solutions trop coûteuses ne sont pas acceptables pour l'industrie, et par conséquent les processeurs modernes soit reposent sur des techniques hybrides, soit sont plus axés sur des techniques de redondance d'information. Les solutions à faible coût disponibles ne disposent pas de capacité de détection d'erreur rapide, et il y a un réel besoin à développer une méthodologie alternative de conception tolérante aux fautes qui puisse combiner une détection d'erreurs rapide et des coûts faibles en surface ou en consommation.

Dans les sections suivantes, différentes méthodes d'évaluation du processeur seront discutées. Nous nous sommes intéressés aux méthodologies contre les fautes transitoires reposant sur l'injection de fautes basée sur la simulation.

II. ÉTUDE QUALITATIVE ET QUANTITATIVE

Chapitre 3

Méthodologie de conception et spécifications du modèle

Avec l'évolution des technologies de fabrication modernes, le risque est grandissant que les fautes transitoires soient de plus en plus fréquentes à l'avenir [FGAD10]. Puisque cette menace sur la fiabilité est susceptible d'affecter le vaste marché informatique, les solutions traditionnelles mettant en œuvre une redondance excessive sont trop chères en termes de surface, de consommation et de performance [BBV+05, SHLR+09]. La recherche sur la conception de systèmes tolérants aux fautes ayant un surcoût matériel minimal a gagné de l'importance dans ces dernières années.

Les précédentes recherches dans ce domaine visaient uniquement à atteindre un haut niveau de fiabilité avec une dégradation minimale des performances, et peu de considération était portée sur les solutions matérielles à faible coût. Par conséquent, la fiabilité est principalement atteinte grâce à des solutions coûteuses comme la réplication du matériel. Les processeurs tolérants aux fautes décrits dans la littérature comme Stratus, Léon FT, Sun SPARC FT, ou IBM S/390, employaient des solutions de redondance matérielle, DMR ou TMR.

Les solutions disponibles impliquent souvent des pénalités importantes en surface, coût, ou en performance, et sont incapables de tolérer efficacement les fautes [PIEP09]. Ils ne peuvent donc pas satisfaire les besoins des applications industrielles courantes. Certaines techniques de redondance temporelle autorisent des surcoûts matériels minimaux, mais génèrent d'importants surcoûts en temps qui limitent la performance globale. D'autre part, la réplication du matériel est plus rapide, mais avec une augmentation du coût et de la consommation. Des techniques d'optimisation efficaces sont donc nécessaires pour répondre aux contraintes de temps et de coût dans le contexte des systèmes tolérants aux fautes. Par conséquent, nous proposons dans ce travail une méthodologie de conception de processeurs présentant un compromis entre la protection et le surcoût en surface et en temps. La section suivante explore la méthodologie proposée.

3.1 Méthodologie

Nous voulons concevoir un processeur tolérant aux fautes ayant une latence de détection d'erreur minimale et un surcoût matériel bas. Dans cette situation, le défi est de trouver un compromis entre la protection et le surcoût matériel. La détection et la correction matérielles sont rapides mais avec un surcoût matériel élevé. D'autre part, la détection et la correction logicielles sont plus lentes mais induisent un surcoût matériel bas.

Par conséquent, pour une détection rapide des erreurs nous avons choisi d'implanter une CED matérielle, pour permettre une détection d'erreur avant qu'elle n'atteigne les limites du système et ne cause une défaillance catastrophique. D'autre part, pour économiser du matériel nous avons toléré une pénalité temporelle supplémentaire dans la correction d'erreur. Par conséquent, le mécanisme de *rollback* logiciel peut être choisi pour le recouvrement d'erreurs.

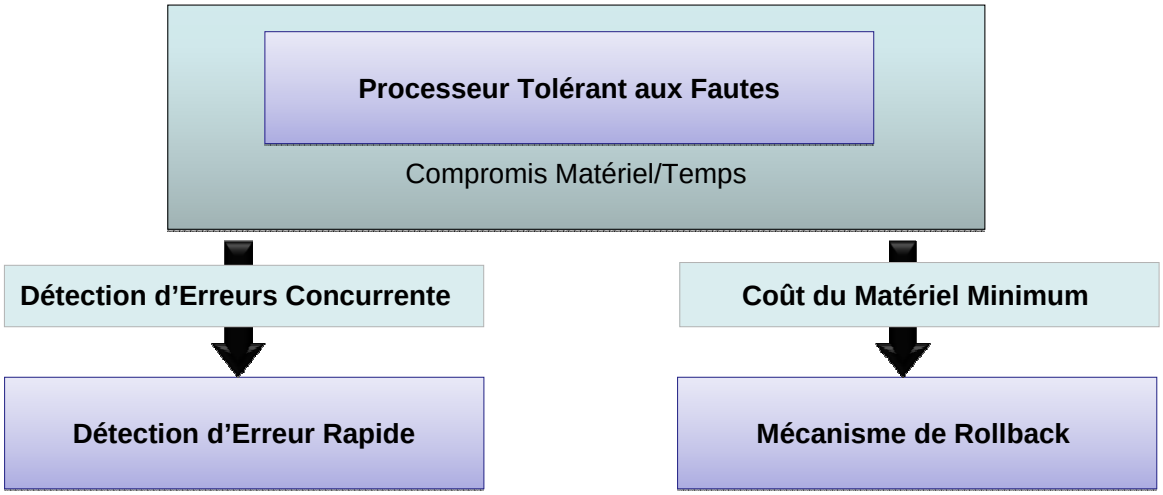


FIGURE 3.1 – Méthodologie proposée

La méthodologie de conception conjointe matériel – le logiciel le qui en résulte (voir figure 3.1) doit permettre de détecter les erreurs dès qu'elles se produisent et de démarrer immédiatement des stratégies de recouvrement des erreurs afin d'éviter leur propagation à travers l'ensemble du système. Dans la section suivante, nous allons discuter de la CED la plus appropriée et des mécanismes de recouvrement pour le scénario ci-dessus.

3.1.1 Détection d'erreurs concurrente : codes de parité

L'implantation de la CED nécessite généralement un surcoût matériel. L'une des approches de CED les plus directes et couramment utilisées est la DMR. Théoriquement, elle peut détecter 100 % des erreurs (sauf les erreurs simultanées dans les deux modules et à l'exception des erreurs dans le comparateur) [MS07]. Cependant, cette technique impose un surcoût en surface supérieure à 200 %. La décision pour le choix de la stratégie de vérification sera guidée par un compromis entre le taux de couverture et un surcoût acceptable.

Des solutions rentables sont l'objet de recherches avancées sur la détection d'erreurs. Les EDC ont des surcoûts en surface plus faibles [Pat10], et ils sont souvent considérés suffisants pour les processeurs non critiques [MS07]. Parmi les EDC, nous allons employer les codes les plus simples, car notre objectif est de montrer la faisabilité de notre approche. Une fois que la méthode globale aura montré des résultats intéressants, alors nous pourrons employer des codes plus robustes avec une meilleure couverture des erreurs.

Les codes de parité sont les plus simples et les moins chers des EDC connus. Il permettent la détection d'erreur en comptant le nombre de bits impairs et nécessitent des circuits supplémentaires pour contrôler la génération des bits et vérifier la parité de sortie. Leur surcoût matériel est beaucoup plus faible que celle de l'approche DMR. Ils peuvent être employés pour protéger les registres, les bus de données, la mémoire RAM et les circuits logiques [Pie06].

L'inconvénient est l'absence de détection des erreurs à multiplicité paire, qui affectent un nombre de bits multiple de deux. L'exemple de la file de registres 8×8 bits de la figure 3.2 illustre ce fait. Les erreurs dans les registres 1 et 3 peuvent être détectées par le contrôle de parité. Cependant, les erreurs dans les registres 2 et 5 ne sont pas détectées.

Les codes de parité n'ont pas besoin d'un circuit complexe pour l'encodage et le décodage. Ils comportent un nombre très faible de portes pour la vérification enligne. Par ailleurs, en cas d'erreurs *soft* où une erreur est aléatoire dans le temps et l'espace, la probabilité d'erreurs multiples dans un même cycle d'horloge est extrêmement faible. Par conséquent, dans ce scénario, une approche peu coûteuse comme la détection d'erreur de parité peut suffire [Gha11].

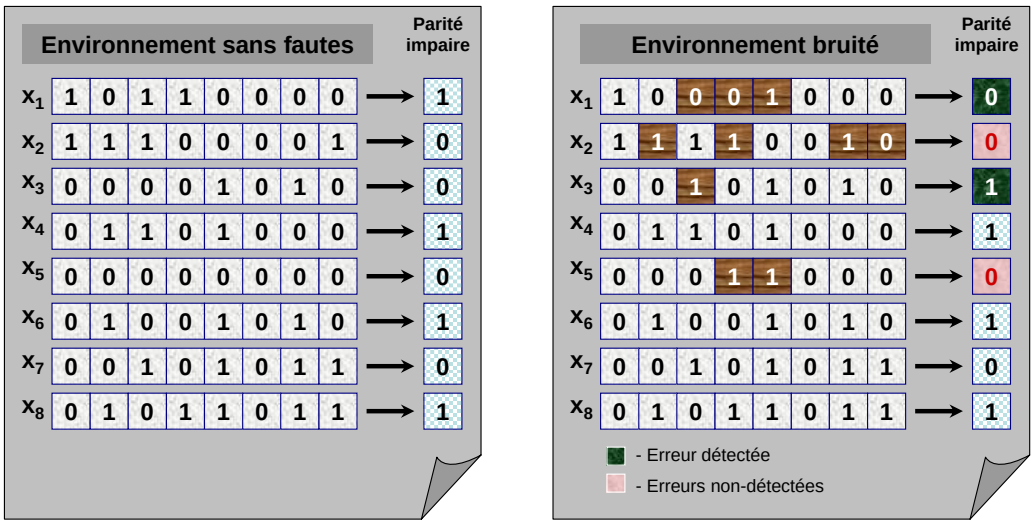


FIGURE 3.2 – Limitation du contrôle de parité

Lisboa [LC08] a employé une approche similaire ; il utilise une technique de parité standard basée sur la détection des erreurs dans les circuits combinatoires à sortie unique. Dans ce travail, un deuxième circuit est utilisé qui génère un signal de sortie supplémentaire, appelé bit de contrôle, et deux circuits pour la vérification de parité des entrées et sorties reposent sur des portes XOR à faible surface pour détecter les erreurs *soft*.

3.1.2 Recouvrement d'erreur : le *rollback*

Pour obtenir un surcoût matériel minimal, la correction d'erreur logicielle sera utile. Une des solutions les plus évidentes est le masquage de faute basé sur la redondance temporelle (TMR logicielle). Elle permet un surcoût matériel faible, mais nécessite trois fois plus de temps pour effectuer les calculs, et de plus ne correspond pas au matériel déjà choisi pour la CED. Une approche alternative peut être basée sur le *rollback* logiciel, grâce auquel les fautes transitoires peuvent être contrées en répétant les opérations de manière contrôlée en réutilisant le même matériel [RRTV02].

Le *rollback* permet de combattre les erreurs en retournant à l'étape de calcul précédant l'apparition de la faute [MG09]. C'est une technique qui permet à un système de tolérer une défaillance par une sauvegarde périodique de l'état du système de manière à ce que – si une erreur est détectée – le retour au *checkpoint* précédent permettra de recouvrer l'erreur [JPS08]. Il peut être un bon candidat pour les situations où des retards dus au recouvrement sont acceptables. Ainsi, le principe du *rollback* peut constituer une approche efficace pour la récupération d'erreur conjointement avec la CED [BP02, EAWJ02].

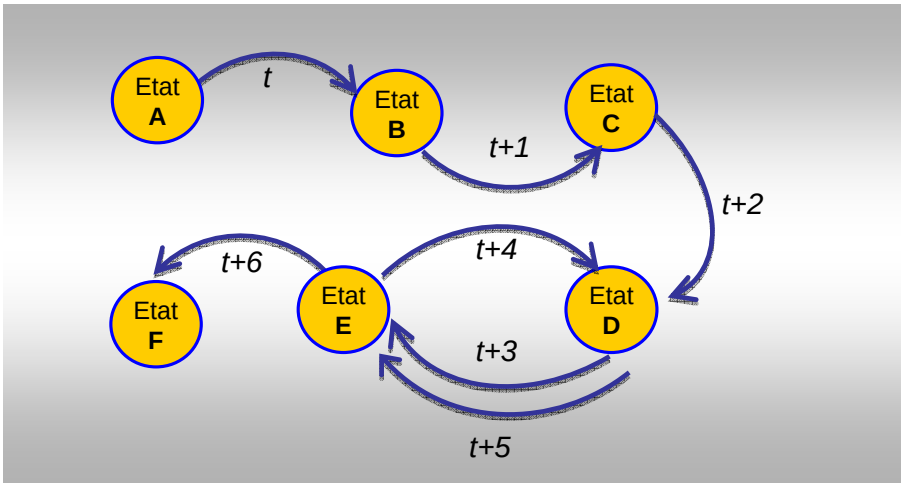


FIGURE 3.3 – Exécution du *rollback*

Notre stratégie d'implantation du recouvrement d'erreur est basée sur le *rollback*, une technique couramment mise en œuvre dans les systèmes embarqués temps-réel [KKB07]. Elle repose sur le fonctionnement suivant (cf figure 3.3) :

- l'exécution du programme (ou du *thread*) est divisée en séquences de longueur maximale fixe ;
- chaque séquence doit être complètement exécutée sans qu'aucune erreur ne soit détectée pour être validée ;
- les données générées lors d'une séquence défectueuse doivent être rejetées et l'exécution recommencée depuis le début de la séquence défectueuse.

Si une erreur apparaît dans les séquences d'instructions suivantes, les registres du processeur peuvent être remis à jour avec leur contenu précédemment sauvegardé. Comme le montre la figure 3.4,

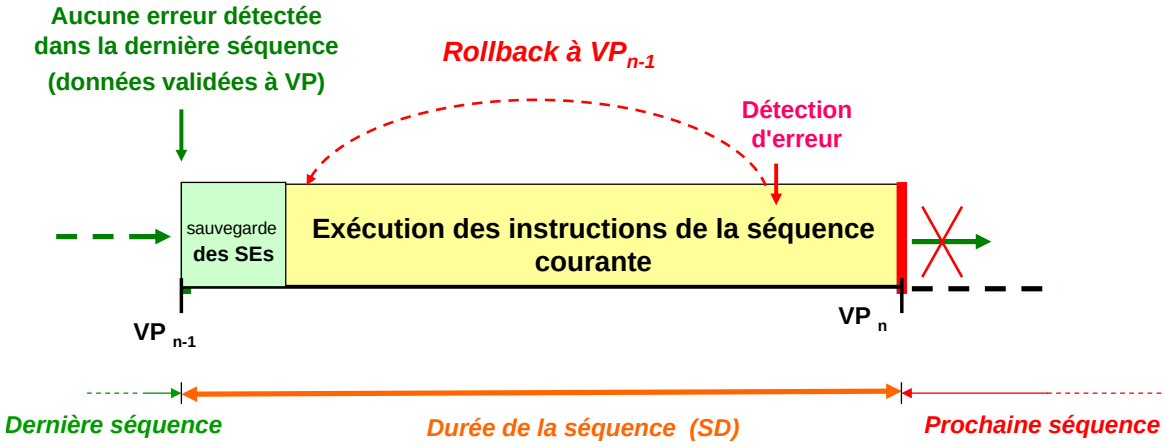


FIGURE 3.4 – Erreur détectée lors d'une séquence d'instruction et appel du *rollback*

lorsqu'une erreur est détectée lors de l'exécution des instructions, le mécanisme de *rollback* est appelé et l'exécution de la même séquence recommence sur la base de l'état sauvegardé lors de la validation précédente, effectuée au « point de validation » (VP : *validation point*). Chaque VP intervient après un nombre d'instructions fixe. D'autre part, comme illustré par la figure 3.5, si aucune erreur n'est trouvée pendant toute la durée de la séquence (SD : *sequence duration*), toutes les données écrites au cours de cette séquence sont validées lors du VP.

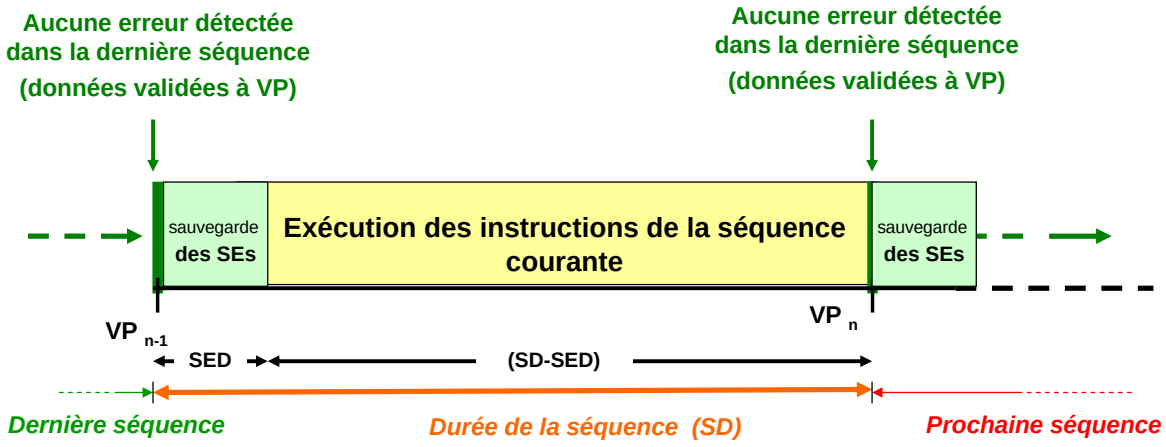


FIGURE 3.5 – Pas d'erreur détectée lors de la séquence

La SD représente la longueur de séquence complète, incluant le temps pris pour sauvegarder les éléments sensibles (SE : *State determining Element*), ainsi que la durée d'exécution des instructions actives. Dans le reste de ce travail, les états internes seront appelés SE. Le temps minimum pour charger les SE sera noté SED (cf figure 3.5). Le ratio de la durée active de la séquence sur la durée totale sera donc (SD-SED)/SD, et le ratio du temps de chargement des SE sur la durée totale sera SED/SD. Pour SD=10, et en supposant que le programme comporte 10000 instructions et en négligeant la possibilité d'apparition d'une erreur, alors il y aura environ 1000 chargement des SE alors que si on fixe SD=100 les SE seront chargés environ 100 fois. Cela signifique la pénalité due au chargement des SE

sera dix fois moindre pour SD=100 que pour SD=10. En résumé, des durées de séquence plus longues permettent une exécution plus rapide du programme.

D'autre part, si la probabilité d'apparition d'erreur n'est pas ignorée, alors la pénalité temporelle due à la réexécution des instructions peut varier avec la longueur de la séquence. À des taux d'injection d'erreur élevés, les séquences les plus courtes seront un compromis efficace car la probabilité de séquences non valides sera plus élevée pour les plus grandes valeurs de SD, et inversement. Ce point sera plus abondamment discuté aux chapitres 5 et 6.

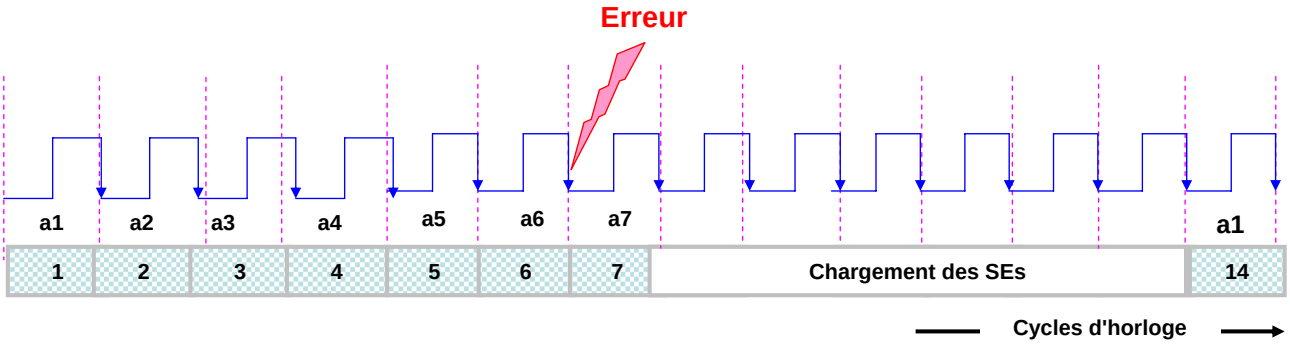


FIGURE 3.6 – Pénalité temporelle due au *rollback*

Pour illustrer la pénalité temporelle en cas de détection d'erreur, si une erreur est détectée lors du cycle 'a7' (cf figure 3.6), le *rollback* est déclenché, le processeur recharge les SE et réexécute toutes les instructions de la séquence en cours. Par conséquent, cela nécessite des cycles d'horloge supplémentaires pour réexécuter la séquence. De plus, le retard sera plus élevé si SD est grande (plus d'instructions dans la séquence).

3.2 Limitations

Un système reposant sur le recouvrement ne peut pas communiquer des données à un environnement en temps-réel, à moins que ces données ne soient garanties sans erreur. Si une donnée erronée est transmise aux périphériques, alors elle ne peut pas être récupérée et peut résulter en défaillance catastrophique. C'est un problème fondamental du recouvrement par *rollback* et a été discuté dans [NMGT06]. L'approche courante consiste à attendre que les données soient validées.

Dans la méthodologie présentée ici, les événements de sortie peuvent être traités avec un retard corresponsant à SD. Cependant, ce retard peut imposer des contraintes de temps-réel. La gestion des événements et les communications temps-réel ne constituent pas l'objet de ce travail, mais cet aspect mérite d'être abordé dans les travaux futurs.

3.3 Hypothèse

Parmi d'autres hypothèses sous-jacentes, nous supposons que le cœur du processeur est relié à une mémoire sûre (DM) dans laquelle les données sont supposées être conservées en toute sécurité sans risque de corruption. Selon cette hypothèse, toutes les erreurs internes produites en mémoire sont détectées et corrigées par la mémoire elle-même. Par conséquent, la mémoire est un lieu de stockage sûr, mais elle doit être protégée des erreurs provenant de l'extérieur, ce qui signifie que seules les données valides peuvent y être écrites.

Beaucoup de travaux ont été consacrés par le passé à la protection de la mémoire [MS06, Hsi10] ce qui rend cette hypothèse valide.

3.4 Défis de conception

Le choix d'un mécanisme de détection d'erreur basé sur la détection concurrente et le recouvrement basé sur le *rollback* ne sont pas suffisants pour atteindre nos objectifs de conception. Une implantation efficace du scénario ci-dessus peut être réalisée en faisant des choix appropriés, particulièrement en ce qui concerne l'architecture du processeur. Ces choix de conception doivent permettre d'améliorer la fiabilité, le coût et la performance globale. Dans la section suivante, nous allons analyser quelques caractéristiques principales nécessaires pour une bonne implantation du scénario ci-dessus.

3.4.1 Défi n°1 : nécessité d'un processeur autocontrôlé

Le choix de l'architecture de processeur est la première étape de l'implantation du processeur tolérant aux fautes car toutes les architectures de processeurs ne présentent pas la même aptitude à répondre aux contraintes liées à ce contexte. Pour une bonne implantation, nous devons déterminer les caractéristiques clefs du processeur.

- *matériel minimal :* nous souhaitons concevoir un processeur tolérant aux fautes qui possède l'empreinte matérielle minimale, ce qui permet de réduire les risques de contamination, car plus la surface exposée à l'environnement est grande, plus le système est sujet aux erreurs. En raison d'une surface plus faible, une architecture plus efficace peut être fabriquée sur une puce de silicium plus petite, et le rendement sera d'autant plus élevé [TM95].

- *un minimum d'états internes à vérifier et à sauvegarder :* (i) avec la détection d'erreurs concurrente, le surcoût matériel nécessaire pour vérifier simultanément l'ensemble des états internes peut être assez important. Le processeur ayant un nombre réduit d'états internes, cela permet de réduire le surcoût du matériel. (ii) Le recouvrement par *rollback* nécessite de sauvegarder périodiquement les états internes, ce qui implique une pénalité temporelle qui peut être réduite avec un nombre réduit d'états internes.

Les processeurs de classe RISC (*Reduce Instruction Set Computers*), couramment utilisés, ont une grande quantité de registres et ne sont pas adaptés à la méthodologie proposée pour les raisons suivantes :

(a) plus de registres signifie des CED plus coûteux ;

(b) plus de registres implique plus de temps pour sauvegarder périodiquement leur contenu ;

(c) un grand nombre de registres nécessite un plus grand nombre de bits d'instruction pour préciser les registres adressés, ce qui signifie un code moins dense ;

(d) Les registres internes au CPU sont plus chers que des emplacements mémoire externes.

D'autre part, le paradigme de conception CISC (*Complex Instruction Set Computers*) nécessite une architecture de contrôle complexe. Cela compliquerait la méthodologie d'implantation globale. Les exigences en mémoire sont également plus élevées. En outre, cela augmente la probabilité d'erreurs de conception.

En résumé, nous ne pouvons pas compter sur les architectures de processeurs classiques (RISC ou CISC). Notre choix se portera donc sur une architecture de processeur simple ayant un minimum d'états internes. Cela permettra de réduire les pénalités en surface et en temps, et de rendre l'architecture plus robuste contre les perturbations externes. D'autre part, une complexité minimale de l'architecture permet une meilleure utilisation des ressources silicium disponibles sur la puce.

3.4.2 Défi n°2 : stockage temporaire nécessaire – journal matériel

Si un processeur autocontrôlé est connecté directement à la mémoire (cf figure 3.7), il est nécessaire de gérer les données validées (VD, écrites lors de la séquence précédemment validée) ainsi que les données non validées (UVD, en cours de traitement dans la séquence actuelle) avant de les transmettre à la mémoire, ce qui va induire une pénalité en temps additionnelle.

Dans ce cas, on utilise généralement une mémoire paginée (une page par séquence), dans laquelle les pages non validées sont séparées des pages validées afin de gérer le *rollback*. Si une erreur est détectée dans la séquence d'instructions en cours d'exécution, la page correspondante est éliminée et la page précédente doit être restaurée. C'est une approche lente qui nécessite des pointeurs supplémentaires pour gérer les pages ; ces pointeurs peuvent être soit des registres dédiés (plus rapide) ou des variables dédiées (plus lent et plus risqué).

Par ailleurs, il y a un risque supplémentaire concernant la corruption de ces pointeurs qui peut conduire à perdre l'information sur la nature des pages, validées ou non validées. La mémoire serait susceptible de contenir des données non sûres, ce qui est en violation avec l'hypothèse de base. Si une grande quantité de données est copiée entre deux pages, ou entre une page et la mémoire, cela nécessitera beaucoup de temps. Par ailleurs, le système nécessiterait une mémoire plus importante pour sauvegarder séparément les données validées et celles non validées.

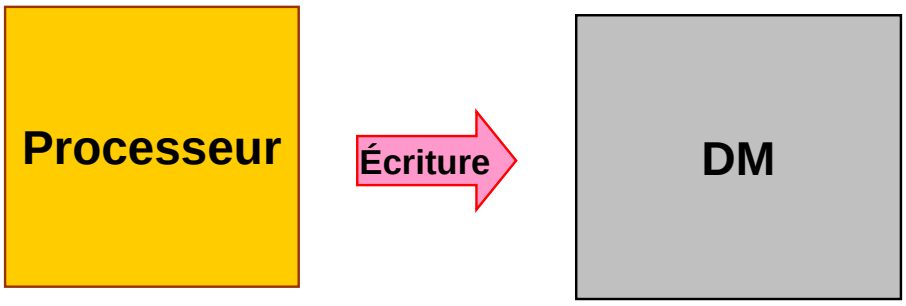


FIGURE 3.7 – Données non sûres écrites vers la mémoire fiable (DM)

Une autre approche pour simplifier le scénario ci-dessus consiste à stocker temporairement les données entre le processeur et la mémoire sûre. Cela peut réduire considérablement la pénalité en temps ainsi que – dans une certaine mesure – le risque d'erreur. Par ailleurs, cela permettra de simplifier la sauvegarde périodique des données, et seules les données validées seront transférés à la DM.

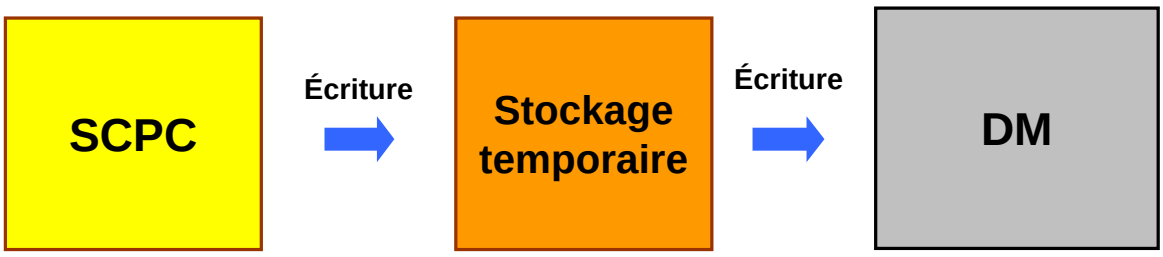


FIGURE 3.8 – Données stockées temporairement avant leur écriture en mémoire

L'idée fondamentale est d'implanter des contrôleurs matériels dans le chemin entre le processeur et la mémoire pour contrôler le flux de données et empêcher les données non sûres de parvenir à la mémoire (comme suggéré dans la figure 3.8). Ceci peut être réalisé en écrivant les données non sûres – c'est à dire non encore validées – dans un endroit temporaire avant de les transférer à la mémoire après que la validation de séquence a eu lieu. Le SCPC peut détecter les erreurs et ré-exécuter les instructions depuis le dernier état sûr (en cas de détection d'erreur). De cette façon, les erreurs externes provenant de l'environnement ou du processeur ne pourront corrompre la mémoire (comme le montre la figure 3.8). L'idée sous-jacente au mécanisme de journalisation est d'éviter que les données non sûres ne parviennent à la mémoire, et de permettre un recouvrement aisé des situations de faute. Il a donc un besoin d'un endroit temporaire (appelé SCHJ – *Self Checking Hardware Journal*, dans les chapitres suivants) pour empêcher les erreurs de se propager à la mémoire.

Nécessité d'un journal matériel autocontrôlé

Les données stockées à l'intérieur de cet emplacement temporaire peuvent également être corrompues par des fautes transitoires telles que les SEU (cf figure 3.9). Par conséquent, il doit y avoir un mécanisme de détection et de correction d'erreur afin d'assurer un fonctionnement fiable de ce stockage temporaire des données.

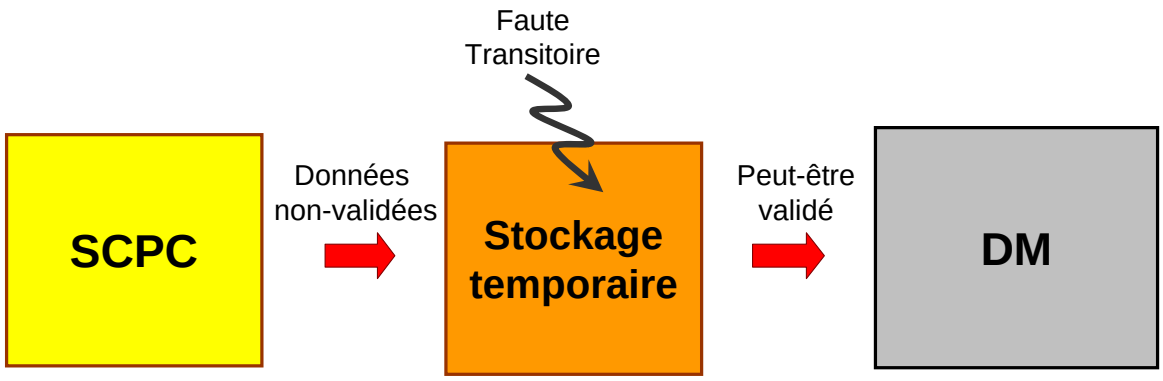


FIGURE 3.9 – Corruption des données dans l'espace de stockage temporaire

Supposons que nous ayons écrit une donnée dans le journal au moment 't', et qu'aucune erreur ne soit détectée pendant la séquence jusqu'au VP, avec des données prêtes à être transférées à la mémoire. Est-ce cette donnée est fiable ? Non, parce que les données sont restées dans le journal pendant un temps 'tx' et la possibilité d'apparition de fautes ne peut être ignorée pendant cet intervalle de temps 'tx'. Ainsi, un mécanisme d'autocontrôle est nécessaire pour détecter les erreurs dans le journal, et donc empêcher la contamination de la mémoire par les données, comme indiqué dans la figure 3.10. Ce mécanisme fera du journal un lieu de stockage temporaire sûr.

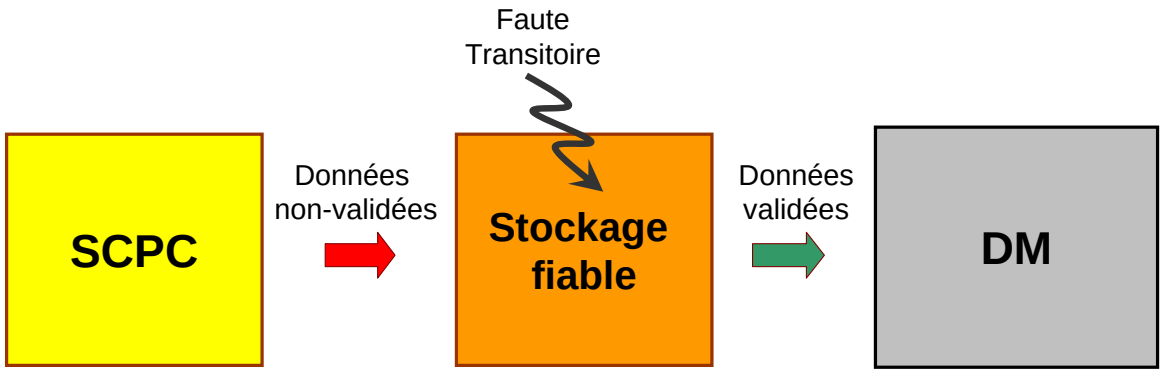


FIGURE 3.10 – Protection de la mémoire contre une contamination.

Stockage séparé des données validées et non-validées

Les données initialement écrites dans l'emplacement temporaire sont non validées, et si aucune erreur ne survient lors de la séquence en cours, alors au point de validation suivant les données seront validées. À tout instant, à l'intérieur du stockage temporaire, il existe deux types de données, l'un nommé « données non validées », et l'autre « données validées ». Par conséquent, le journal doit comporter deux parties différentes : une pour stocker les données validées, et l'autres pour celles qui ne l'ont pas encore été. En outre, cette séparation physique entre les types de données facilitera le transfert des données validées vers la mémoire.

3.4.3 Défi n°3 : interface entre processeur et mémoire

La performance globale du processeur tolérant aux fautes peut être limitée par l'absence d'une interface efficace entre le processeur, l'endroit de stockage temporaire, et la mémoire. Comme dans la plupart des processeurs, la majorité des instructions consiste en la lecture ou l'écriture de et vers la mémoire. La performance globale est affectée si le chemin critique est trop long, ou si plus d'un cycle d'horloge est nécessaire pour lire et écrire les données. Dans notre cas, la situation est plus délicate à gérer car il existe un stockage temporaire des données entre le processeur et la mémoire : il est nécessaire de concevoir une interface évoluée pour empêcher la propagation des erreurs vers la mémoire. Cette interface doit fournir une interconnexion efficace entre les modules.

Dans ce scénario, deux types d'interfaçage sont possibles : soit le processeur communique avec la mémoire via un journal, soit le processeur communique en parallèle avec le journal et la mémoire. Le défi est d'évaluer les deux modèles de processeur du point de vue de la fiabilité et de la dégradation des performances afin de choisir le plus approprié.

3.4.4 Défi n°4 : durée de séquence optimale pour implanter efficacement le mécanisme de *rollback*

L'objectif de la technique du *rollback* est de restaurer l'état du système (en cas d'erreur) en remplaçant les états actuels avec les états précédemment validés des SE, comme indiqué dans la figure 3.4. Il y a deux facteurs limitant la performance : (i) le temps nécessaire pour sauvegarder périodiquement les SE, et (ii) l'invalidation de la séquence actuelle et la restauration des SE en cas de détection d'erreur. Si nous avons besoin de réduire la pénalité temporelle induite par la restauration des SE, il nous faudra mettre en œuvre de grandes séquences pour que le nombre global de chargements/sauvegardes des SE représente une durée minimale au sein de la durée totale de la séquence SD. D'autre part, pour de longues séquences et sous des taux d'erreurs plus élevés, les chances de valider la séquence diminuent. Par conséquent, le taux de *rollback* augmente, résultant également en une pénalité temporelle, et donc en une dégradation des performances. Il est donc recommandé d'utiliser de grandes séquences avec des faibles taux d'erreur et des séquences courtes avec des taux d'erreur plus élevés.

3.5 Spécifications du modèle et flot de conception global

Le rôle fondamental du Journal est de conserver les nouvelles données générées lors de la séquence en cours d'exécution jusqu'à ce qu'elles puissent être validées à la fin de la séquence. Après la validation de la séquence, ces données peuvent être transférées à la mémoire, sinon elles sont tout simplement rejetées et la séquence en cours peut reprendre depuis le début après restauration des SE sauvegardés dans la mémoire sûre, et qui correspondent à l'état sûr à la fin de la séquence précédente.

Notre stratégie de conception d'un processeur tolérant aux fautes est décomposée en quatre étapes résumées par la figure 3.12. L'étape I résume les spécifications du modèle proposé, comme indiqué

FIGURE 3.11 – Spécifications de conception

dans le schéma de la figure 3.11. Cela inclut l'analyse des besoins de conception comme, dans le cas présent, le SCPC, la DM et le SCHJ, pour empêcher les erreurs de parvenir à la DM. Par ailleurs, l'architecture doit respecter les défis 1 à 4 mentionnés dans la section précédente.

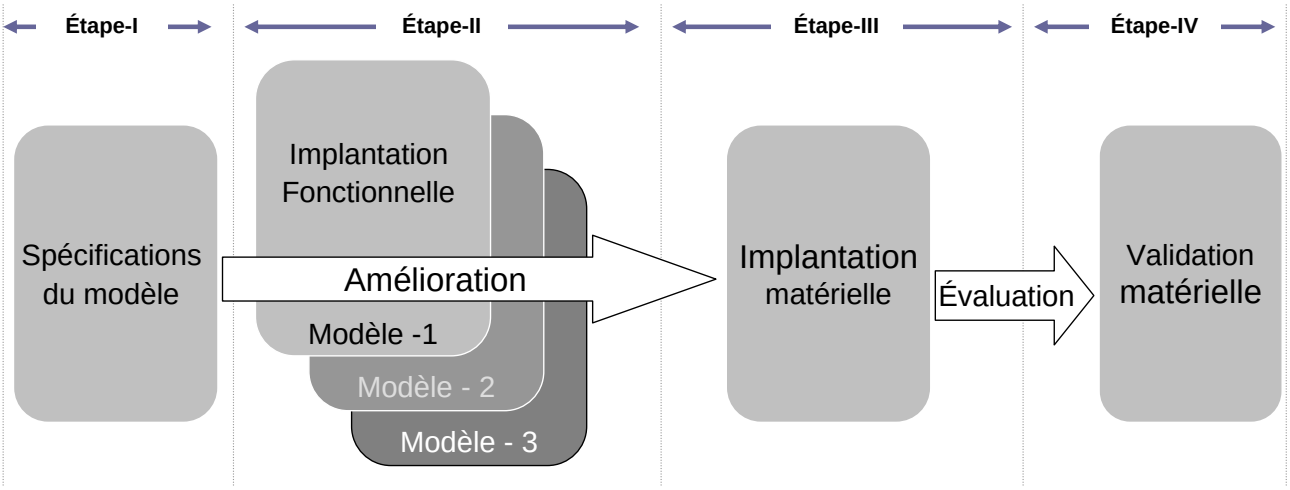


FIGURE 3.12 – Flux de conception global

La deuxième étape permet d'affiner notre stratégie de conception en utilisant diverses implantations fonctionnelles. Elle sera présentée dans la section suivante. L'implantation matérielle, étape III, sera présentée dans les chapitres 4 et 5. Enfin, la quatrième étape porte sur la validation de l'approche globale par l'injection d'erreurs artificielle. Elle sera présentée dans le chapitre 6.

3.6 Implantation fonctionnelle

Cette section est consacrée à l'étape II de la figure 3.12 qui vise à affiner le modèle proposé. Deux connexions sont possibles entre le processeur et la mémoire : (i) Modèle I : le processeur est connecté à la mémoire via le journal et le processeur ne peut pas lire la mémoire en un cycle d'horloge ; (ii) Modèle II : le processeur est connecté au journal et à la mémoire en parallèle et il peut lire simultanément la mémoire et le journal. En conséquence, la fiabilité et les performances globales seront affectées par le choix effectué. Dans cette section, nous allons finaliser l'interface entre la

mémoire et le processeur.

Déterminer quel est le meilleur scénario peut être effectué par le développement des modèles fonctionnels correspondants, puis en comparant les courbes de simulation qui décrivent les cycles d'horloge nécessaires par instruction (*Clock per Instruction*, CPI) par rapport au taux d'injection d'erreur (*Error Injection Rate*, EIR) obtenu par l'injection artificielle d'erreur.

Hypothèses

Afin de simplifier le modèle fonctionnel, les hypothèses suivantes sont posées :

- le cœur du processeur est autocontrôlé ;
- il y a une mémoire fiable attachée au processeur où les données sont conservées sans risque d'erreur ;
- le journal est considéré comme un lieu de stockage de données sûr ;
- toutes les instructions sont supposées être exécutées en un cycle d'horloge ;
- et bien sûr, la réexécution des instructions permet de recouvrer les erreurs *soft*.

Benchmarks

Un ensemble de benchmarks comportant les principales tâches susceptibles de constituer l'application cible a été sélectionné et divisé en trois groupes. Le premier groupe est l'exécution des opérations mémoire (permutation, tri). Le deuxième groupe est représentatif des algorithmes dominés par les opérations arithmétiques. Enfin, le troisième groupe contient les algorithmes orientés contrôle. Toutes les applications ont des exigences de mémoire importante, car chaque fois qu'elles accèdent à la mémoire pour y lire une donnée, elles doivent ensuite y écrire le résultat après l'exécution (ces benchmarks ne sont pas conçus pour évaluer les événements d'entrée-sortie, par conséquent les opérations de lecture et d'écriture ne concernent que la mémoire principale).

(a) *Benchmark Groupe I :* L'algorithme de tri à bulle est considéré comme l'un des algorithmes de tri d'un tableau les plus simples. Il consiste en une boucle d'échanges répétés de paires adjacentes de données dans un tableau en mémoire. L'algorithme se répète jusqu'à ce que tous les éléments du tableau soient dans le bon ordre. Il a été implanté en mode série, dans lequel une seule paire de données peut être examinée à la fois. Il a une complexité en temps de $O(n2)$. Les n passes doivent être effectuées sur tout le tableau, où n est le nombre d'éléments.

(b) *Benchmark Groupe II :* C'est un benchmark de calcul en mémoire qui nécessite l'écriture de données à des adresses proches. Cette version multiplie deux matrices 7×7 en $O(n^k)$ avec $k > 2$. L'exécution est obtenue en implantant la multiplication d'une matrice par un vecteur. Une matrice initiale est enregistrée en mémoire, et son produit avec un vecteur d'entrée est retourné de manière répétitive.

(c) *Benchmark Groupe III :* Le benchmark de contrôle traite les données provenant d'un capteur et préalablement stockées dans la mémoire. Les résultats sont sauvegardés en mémoire pour être ensuite utilisés par les actionneurs. Nous avons choisi des équations logiques et arithmétiques pour traiter les données, car certains systèmes industriels ont besoin de contrôler des actionneurs avec ce type d'équations. Il y a deux hypothèses : (i) les mesures issues des capteurs sont stockées en mémoire ; (ii) les résultats seront envoyés ultérieurement aux actionneurs.

Les équations de contrôle sont :

$$
\begin{aligned}
Y_0 &= A \times [(X_0 + X_1) - (X_2 - X_3)]/[X_4 \times (-X_5)] \\
Y_1 &= \text{ NOT } [(X_6 \text{ OR } X_1) \text{ AND } (X_9 \text{ XOR } X_7)] \text{ AND } [\text{NOT } (X_8)] \\
&\quad \text{if } ((X_8 + X_9) < A) \\
Y_2 &= B \times [(Y_0 + X_1) - (X_9 \times X_8)]/[X_1 - X_5] + \text{ C} \\
&\quad \text{else} \\
Y_2 &= [(Y_0 + X_1) - (X_9 \times X_8)]\,[X_1 - X_5] + \text{ D} \\
Y_3 &= \text{ NOT}[(X_6 \text{ OR } Y_1) AND (X_9 \text{ XOR } X_7)] \text{ AND } [\text{ NOT } (Y_1)]
\end{aligned}
$$

3.6.1 Modèle I

Dans le modèle I présenté par la figure 3.13, le processeur est connecté à la mémoire via une mémoire cache et une paire de journaux. La paire de journaux masque les erreurs afin de ne pas les propager vers la mémoire. L'opération d'écriture par le processeur dans la mémoire est modifiée et réalisée en trois étapes, comme indiqué dans la figure 3.13 : (i) l'opération d'écriture est effectuée simultanément vers le cache et vers le journal non-validé (*Un-Validated Journal*, UVJ) ; (ii) si aucune erreur n'est détectée, au point de validation les données présentes dans le journal non validé sont transférées au journal validé (*Validated Journal*, VJ) qui contient uniquement les données validées ; enfin, (iii) les données validées sont écrites mémoire.

Au point de validation, tous les derniers états sûrs des SE sont conservés et en cours de validation. Comme le montre la figure 3.13, les données sont transférées à partir du journal non validé vers le journal validé, puis enfin vers la mémoire. Si une erreur est détectée lors d'une séquence, le processeur recommence l'exécution des instructions depuis le précédent VP (cf figure 3.4). De cette façon, le système restaure un état fiable et la mémoire reste préservée de l'erreur. Lors de la validation de séquence, les données présentes dans le journal non validé sont validées en effectuant un transfert synchrone depuis le journal non validé vers le journal validé en un seul cycle d'horloge.

Le processeur peut lire directement à partir de la mémoire cache. Un cache associatif est utilisé, où chaque bloc est composé à la fois de l'adresse mémoire et des données correspondantes. L'adresse entrante est comparée simultanément avec toutes les adresses stockées grâce à la logique interne de la mémoire associative, comme illustré par la figure 3.14. Si une correspondance est trouvée, les

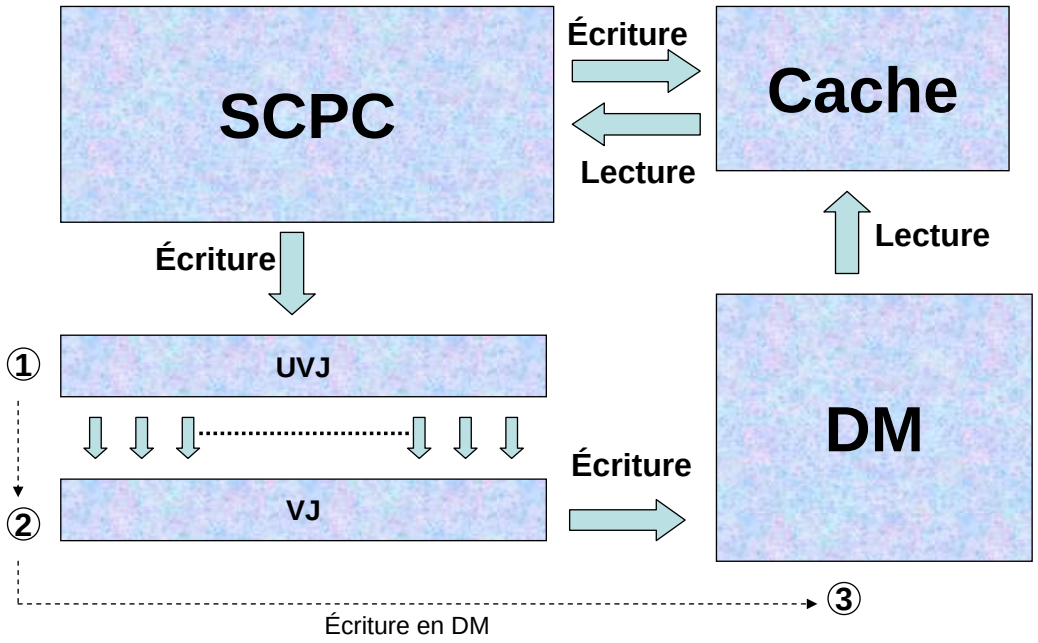


FIGURE 3.13 – Modèle I avec un cache de données et une paire de journaux

données correspondantes sont lues. Sinon, les données nécessaires seront lues depuis la mémoire. Lorsque de nouvelles données sont écrites dans le cache, le contrôleur va d'abord essayer de trouver une adresse disponible correspondante, puis écraser les données présentes à cette même adresse. Si aucune correspondance n'est trouvée, alors les données sont écrites à une nouvelle position avec l'adresse correspondante. L'utilisation de la mémoire associative est rapide, mais aussi coûteuse. Le coût dépend de la taille du cache.

Modèle I : résultats de simulation

Un modèle fonctionnel du processeur autocontrôlé associé au cache et au journal a été développé en C++. L'émulateur se comporte comme une machine virtuelle pour tester les différent modèles de fautes et les techniques de protection avant l'implantation matérielle. En outre, l'émulateur nous permet de valider l'architecture, de calculer les états internes du processeur et la durée d'exécution du programme, et nous aide à calculer le nombre moyen de cycles d'horloge par instruction pour différents types d'accès mémoire.

Les erreurs sont artificiellement injectées dans l'émulateur du processeur (comme illustré par la figure 3.15) puis la performance du modèle de processeur est évaluée. L'objectif de ce dispositif expérimental est d'évaluer l'effet de l'injection d'erreur sur les performances du système. Pour plus de simplicité, le surcoût temporel réel de la sauvegarde périodique des SE est ignoré. Les profils d'injection de fautes considérés sont présentés dans la figure 3.16.

L'émulateur exécute les benchmarks précédemment décrits et qui représentent les applications cibles avec un ensemble de données représentatives. L'entrée de l'émulateur est un fichier hexadécimal classique. Notre critère d'évaluation est le ratio du nombre moyen de cycles d'horloge par

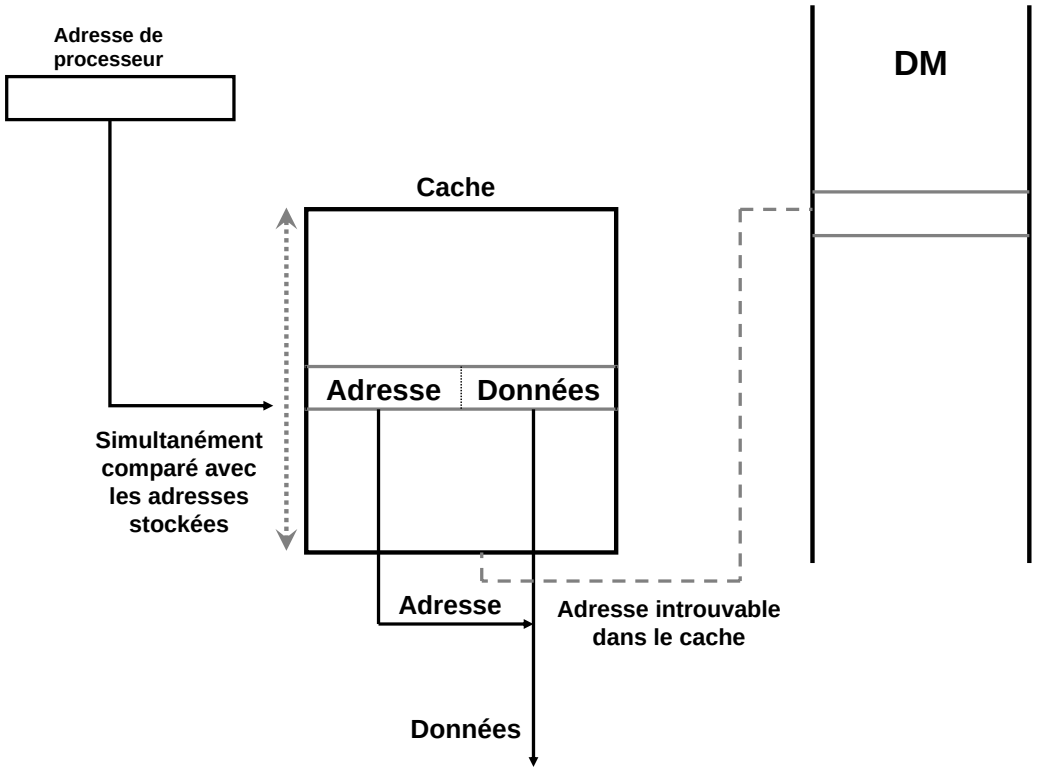


FIGURE 3.14 – Cache associatif

instruction (CPI) sur le taux d'injection d'erreur (EIR). Le but de ces simulations est d'évaluer la dégradation des performances du modèle proposé en présence de taux élevés d'injection d'erreur.

Les figures 3.17, 3.18, et 3.19 présentent les résultats obtenus pour les groupes de benchmarks 1, 2 et 3, pour des durées de séquence SD de 10, 50 et 100 instructions. CPI* représente le nombre de cycles d'horloge par instruction avec injection d'erreur, et CPI correspond au nombre de cycles d'horloge par instruction sans injection erreur. Le ratio CPI*/CPI nous donne le rapport du nombre de cycles additionnels requis lors de la réexécution à cause du *rollback*.

La figure 3.17 montre les résultats de simulation du benchmark de calcul. Dans ce graphique, deux lignes horizontales de référence servent de référence. La ligne verte pointillée du bas correspond à la valeur de CPI*/CPI en l'absence d'erreur, alors que la ligne rouge discontinue du haut représente la limite arbitrairement choisie de $2\times$CPI*/CPI. Trois courbes dans chaque figure correspondent à des séquences de durée 10, 50 et 100 instructions, respectivement. Les courbes sont pratiquement confondues pour des taux d'erreur faibles. Au fur et à mesure que le taux d'erreurs injectées EIR augmente, le ratio CPI*/CPI augmente de façon exponentielle, et est plus important pour des valeurs hautes de SD comme 50 ou 100. Ainsi, en cas de fort taux d'erreurs, le taux de réexécution augmente également, ce qui augmente finalement le ratio CPI*/CPI. Ce modèle donne un meilleur ratio CPI*/CPI pour des taux d'erreurs faibles, mais pour des taux plus élevés le ratio de CPI augmente rapidement en raison de la réexécution des instructions due à la détection d'erreurs ainsi qu'aux cycles d'horloge supplémentaires liés aux défauts de cache. Ces deux problèmes seront abordés dans le modèle II.

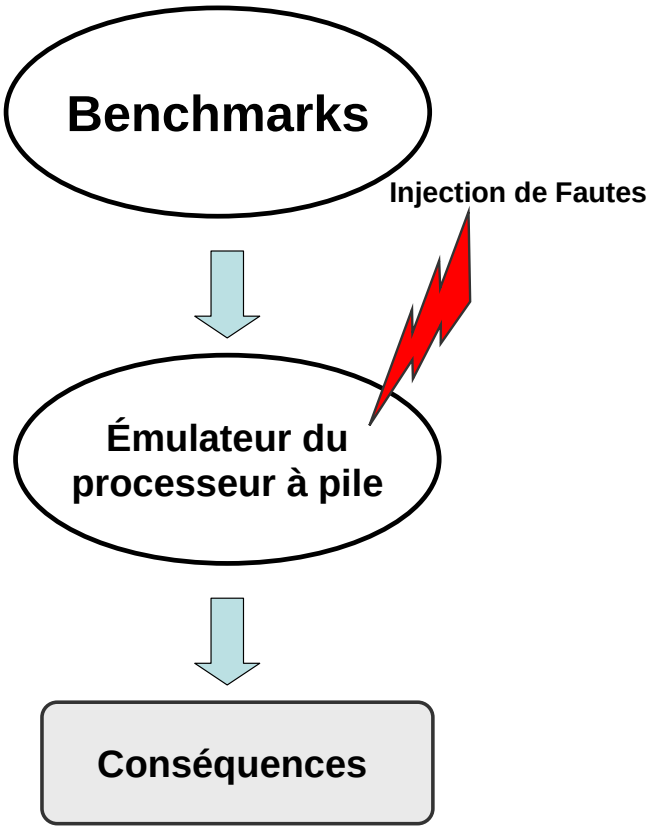


FIGURE 3.15 – Évaluation de la tolérance aux fautes

3.6.2 Modèle II

Le modèle II se compose de trois parties : un cœur de processeur autocontrôlé, le journal, et la mémoire, comme indiqué dans la figure 3.20 (a). L'architecture du journal a été modifiée et comporte deux parties internes : l'une contient les données validées et l'autres les données non validées comme indiqué dans la figure 3.22.

Lorsque le processeur doit effectuer une lecture, il vérifie la présence des données dans le journal et la mémoire simultanément, grâce à un accès parallèle. Un mappage associatif est employé dans le journal, chaque bloc étant composé de l'adresse mémoire et de la données correspondante (comme le montre la figure 3.22). Si un « défaut de journal » (défaut de cache dans le journal) se produit, alors les données requises seront envoyées au processeur dans le même cycle d'horloge. Si les données sont présentes à la fois dans le journal et la mémoire, alors le contrôleur (MUX) va privilégier les données du journal, plus récentes que celles en mémoire (cf figure 3.21).

Afin de permettre la lecture et l'écriture simultanées dans le journal, celui-ci doit avoir deux ports d'adresses. Certaines instructions peuvent nécessiter deux opérations – une lecture et une écriture – en même temps dans le journal. Les données nouvellement écrites sont stockées dans le journal non validé. Si aucune erreur n'est détectée lors de la séquence, alors les données sont validées (comme indiqué dans la figure 3.22) puis transférées au journal validé. D'autre part, si une erreur est détectée, alors toutes les données écrites pendant la séquence sont rejetées (comme illustré par la figure 3.23) et

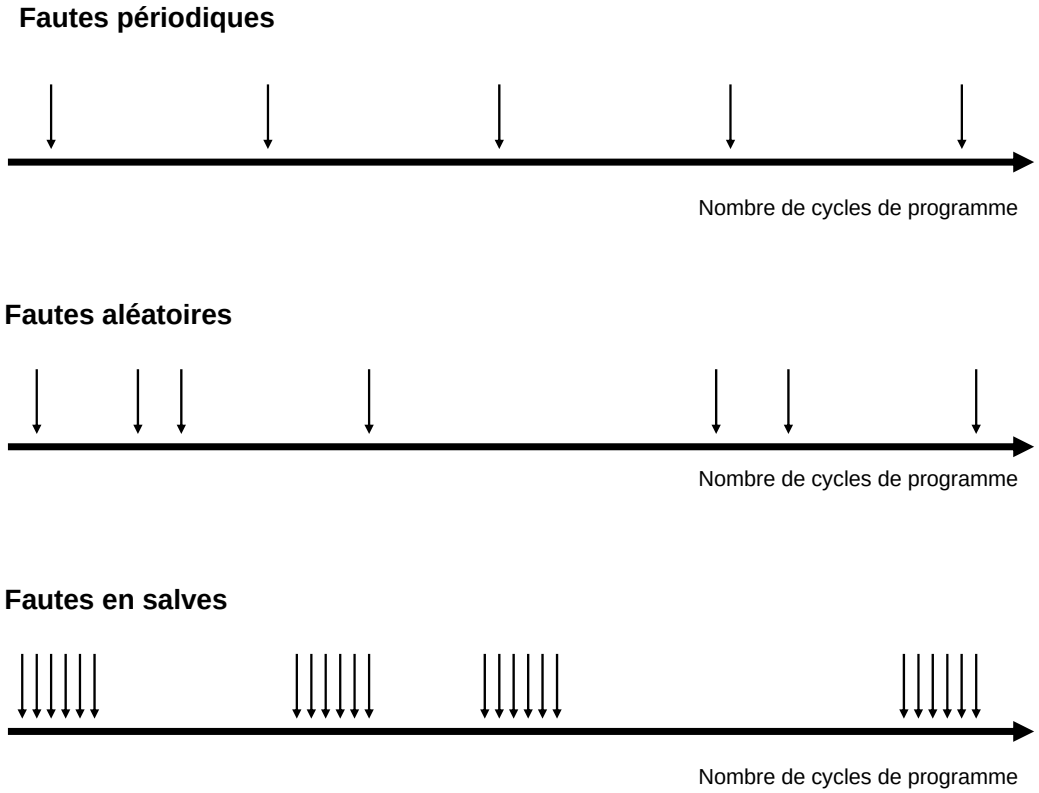


FIGURE 3.16 – Modèles d'erreurs périodiques, aléatoires et en salves

le processeur recommence l'exécution de la séquence en rechargeant le dernier état sûr des SE. Dans la section suivante, nous allons évaluer les performances de cette architecture.

Modèle II : résultats des simulations

Le protocole expérimental reste le même pour le modèle II que pour le modèle I. Nous pouvons observer à partir des courbes de simulation des figures 3.24, 3.25, et 3.26, que ce modèle II est plus efficace que le modèle I, parce que même en présence de taux d'erreurs élevés les ratios CPI*/CPI sont significativement plus faibles qu'avec l'architecture précédente.

À partir des résultats de simulation présentés dans les graphiques 3.24, 3.25, et 3.26, nous pouvons constater que le ratio CPI */CPI est inférieur à celui du modèle I. Dans le graphique 3.24 et pour une durée de séquence SD de 10, lorsque le taux d'injection EIR varie de 2×10^{-4} à 2×10^{-2}, le CPI augmente seulement de $50\ \%$ (sur l'axe Y, le ratio CPI*/CPI est de 1,5), ce qui signifie que le temps d'exécution augmente de seulement $50\ \%$, même si le taux d'injection EIR est 100 fois plus élevé. Cela démontre une bonne performance pour l'architecture proposée. Par exemple, si nous acceptons une augmentation de $50\ \%$ du CPI, avec une SD de 10, il peut se produire 20 erreurs pour 1000 instructions, alors qu'avec une SD de 50 il y aura seulement 6 erreurs pour 1000 instructions. Par ailleurs, la valeur de SD a une incidence directe sur la quantité de mémoire nécessaire dans le journal, et par conséquent sur sa surface.

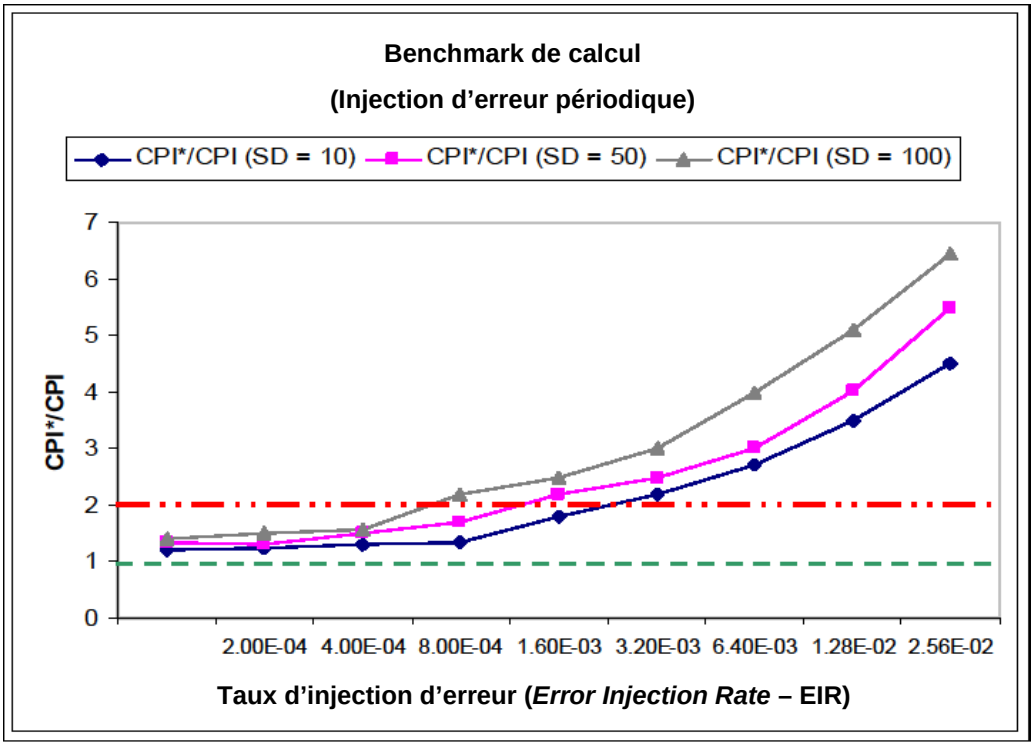


FIGURE 3.17 – Modèle I : CPI additionnels pour les benchmarks du groupe I

3.6.3 Comparaison

La comparaison entre les modèles I et II est résumée dans le tableau 3.1. Dans les deux modèles, l'effet du *rollback* est dominant pour des durées de séquence SD élevées comme 50 ou 100. Par exemple, lorsque le VP se produit après la centaine d'instructions de la séquence si SD=100, les chances d'occurence d'erreurs sont plus importantes, ce qui augmente le ratio CPI*/CPI plus rapidement que par rapport aux durées de séquence SD plus courtes comme 10 ou 50. En outre, avec un grand intervalle entre deux VP consécutifs, l'occurrence plus élevées d'erreurs se traduit par une augmentation du taux de réexécution des instructions.

Du point de vue des performances dans le modèle II, comparativement au modèle I et en raison de l'accès parallèle à la mémoire et au journal pour les opération de lecture, l'efficacité globale du système est augmentée, entraînant une baisse des ratios CPI pour des EIR élevés. Par conséquent, aucun cycle d'horloge n'est perdu si la donnée n'est pas trouvée dans le journal. Ce modèle II offre une meilleure performance que le modèle précédent comme le montrent les graphiques 3.24, 3.25, et 3.26.

Du point de vue de la fiabilité, le modèle II est un meilleur choix parce qu'il génère un surcoût matériel minimale grâce à la présence d'un seul journal par rapport au modèle I qui nécessite une mémoire cache supplémentaire. La plus grande surface exposée à l'environnement du modèle I augmente également les probabilités d'erreurs. Ces deux problèmes, la dégradation des performances à des taux d'injection d'erreurs élevés et la surface nécessaire sur la puce sont mieux résolus par le modèle II que par le modèle I. Par conséquent, nous choisirons le modèle II pour le développement

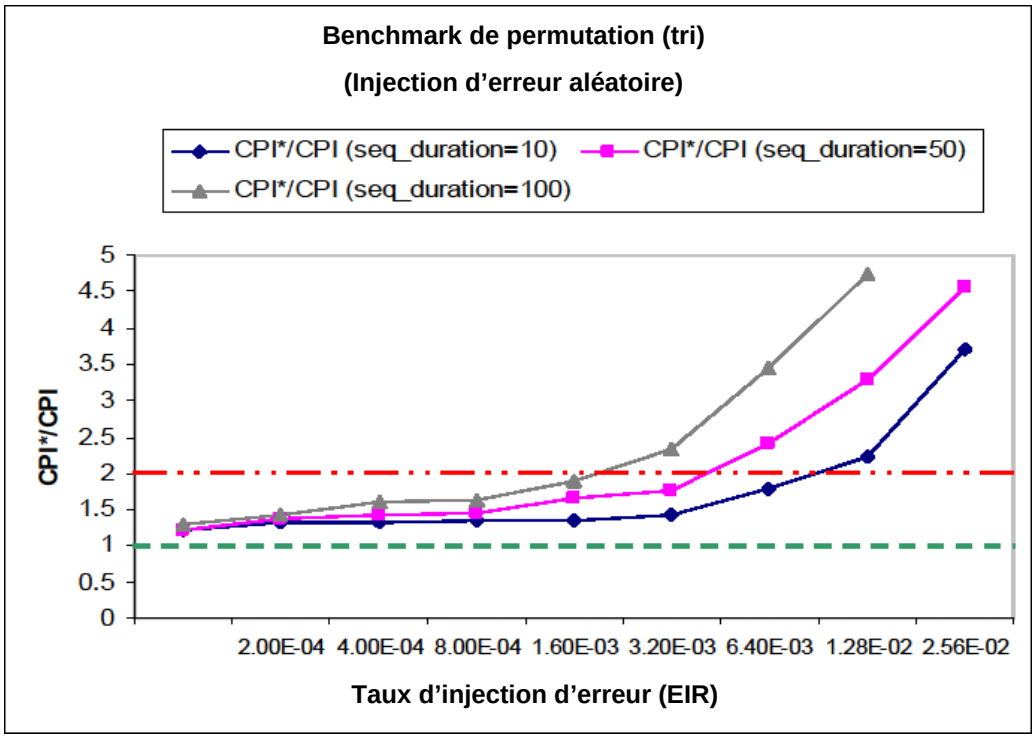


FIGURE 3.18 – Modèle I : CPI additionnels pour les benchmarks du groupe II

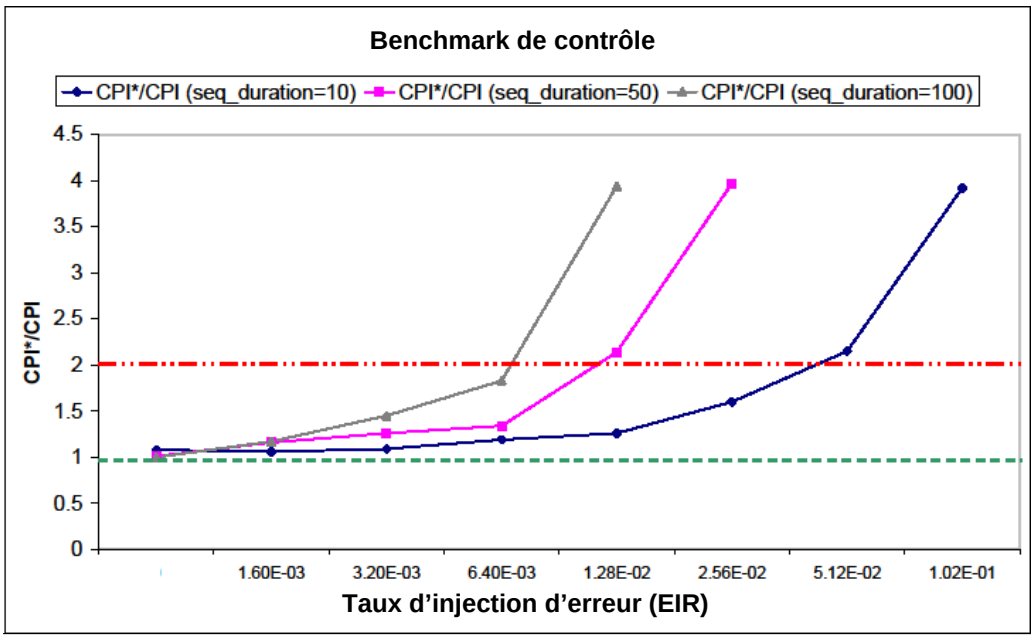


FIGURE 3.19 – Modèle I : CPI additionnels pour les benchmarks du groupe III

de notre processeur. Les résultats obtenus sont assez encourageants pour poursuivre les recherches en nous reposant sur ce modèle. Dans les deux prochains chapitres, nous traiterons de la conception du processeur (chapitre 4) et du journal (chapitre 5).

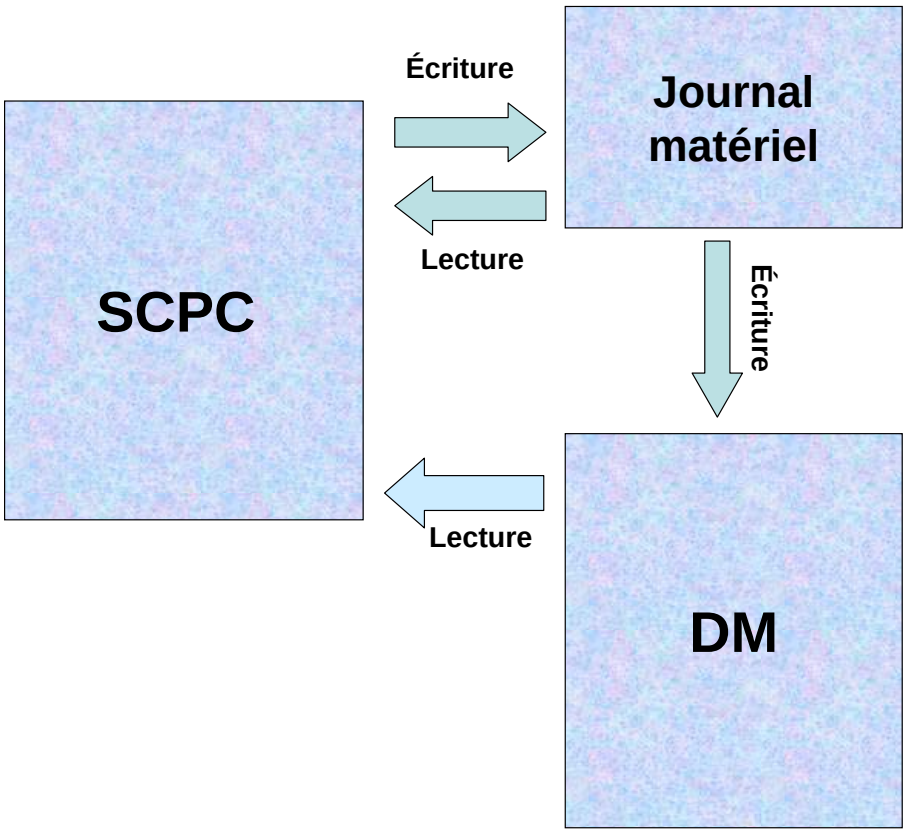


FIGURE 3.20 – Schéma du modèle II

TABLE 3.1 – Comparaison des modèles processeur-mémoire

	Modèle I	Modèle II
Lecture de la mémoire (DM)	Processeur ⇐ Cache ⇐ DM	Processeur ⇐ DM
Lecture du cache/journal	Processeur ⇐ Cache	Processeur ⇐ Journal
Ècriture dans la DM	Processeur ⇒ UVJ ⇒ VJ ⇒ DM	Processeur ⇒ Journal ⇒ DM
Taille requise pour le cache/journal (pour éviter les défauts de cache)	Comparativement plus élevée Cache nécessaire	Pas d'échec de lecture dans le journal en raison de l'accès parallèle
Performance	Moyenne	Bonnes

3.7 Conclusion

Ce chapitre résume une approche alternative de conception d'un processeur tolérant aux fautes. Nous avons présenté une spécification de l'architecture et la méthodologie de conception du paradigme proposé. Il s'agit d'une approche combinée matériel/le logiciel le dans laquelle la détection d'erreur est réalisée concurremment en utilisant les codes de parité, et le *rollback* est utilisé pour recouvrer les erreurs. Les grands avantages de ce scénario sont la possibilité d'avoir un mécanisme de tolérance aux fautes efficace, avec un surcoût matériel et temporel limité. La méthodologie glo-

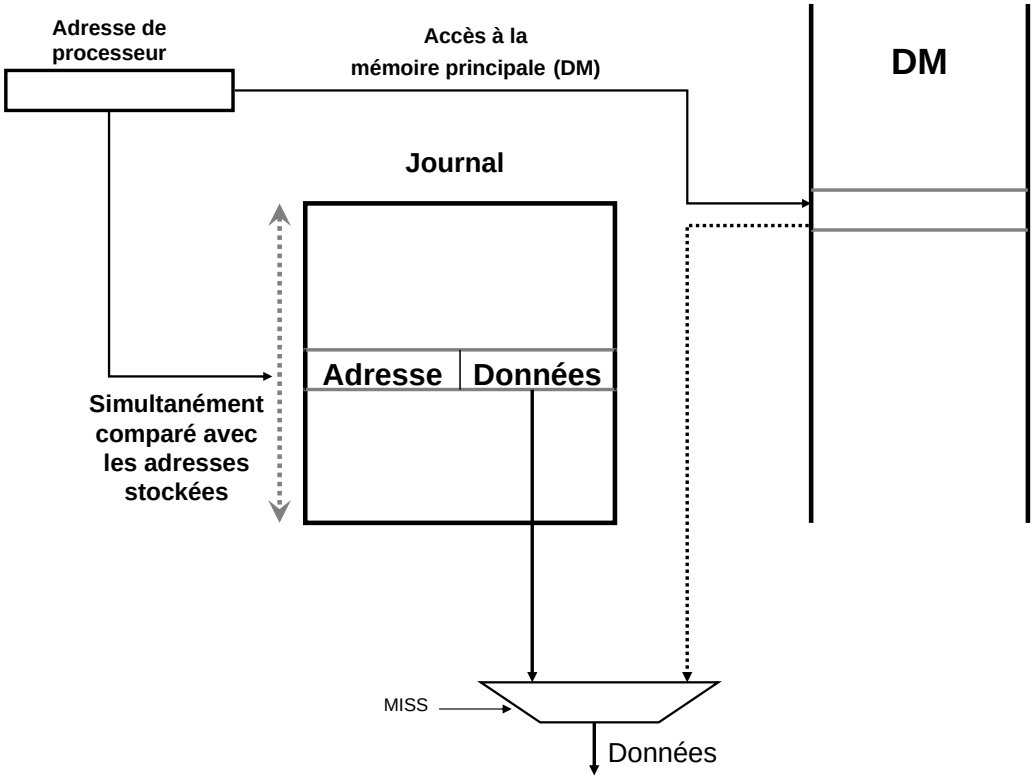


FIGURE 3.21 – Le processeur peut lire simultanément dans le journal et en mémoire

bale peut être efficace si certains défis de conception sont respectés comme le choix d'un processeur approprié avec un minimum d'états internes à charger et à sauvegarder, la conception d'un journal matériel intermédiaire autocontrôlé pour empêcher les erreurs de se propager vers la mémoire sûre, et une durée de séquence raisonnable pour des taux d'erreur donnés.

La dernière partie du chapitre a été consacrée à la définition de l'interface processeur-mémoire. Nous avons proposé deux modèles différents : les modèles I et II. Après comparaison, le modèle II a été sélectionné pour poursuivre le développement en vue d'un modèle VHDL-RTL, parce qu'il offre le meilleur compromis du point de vue de la fiabilité et de la performance. Dans ce modèle, pour l'écriture en mémoire, les données transitent par un stockage temporaire avant d'atteindre la DM, alors que pour la lecture le processeur peut accéder directement à la DM. De cette façon, la mémoire reste préservée de la potentielle propagation d'erreurs provenant du processeur. Dans les prochains chapitres, nous allons développer le modèle VHDL-RTL du processeur tolérant aux fautes.

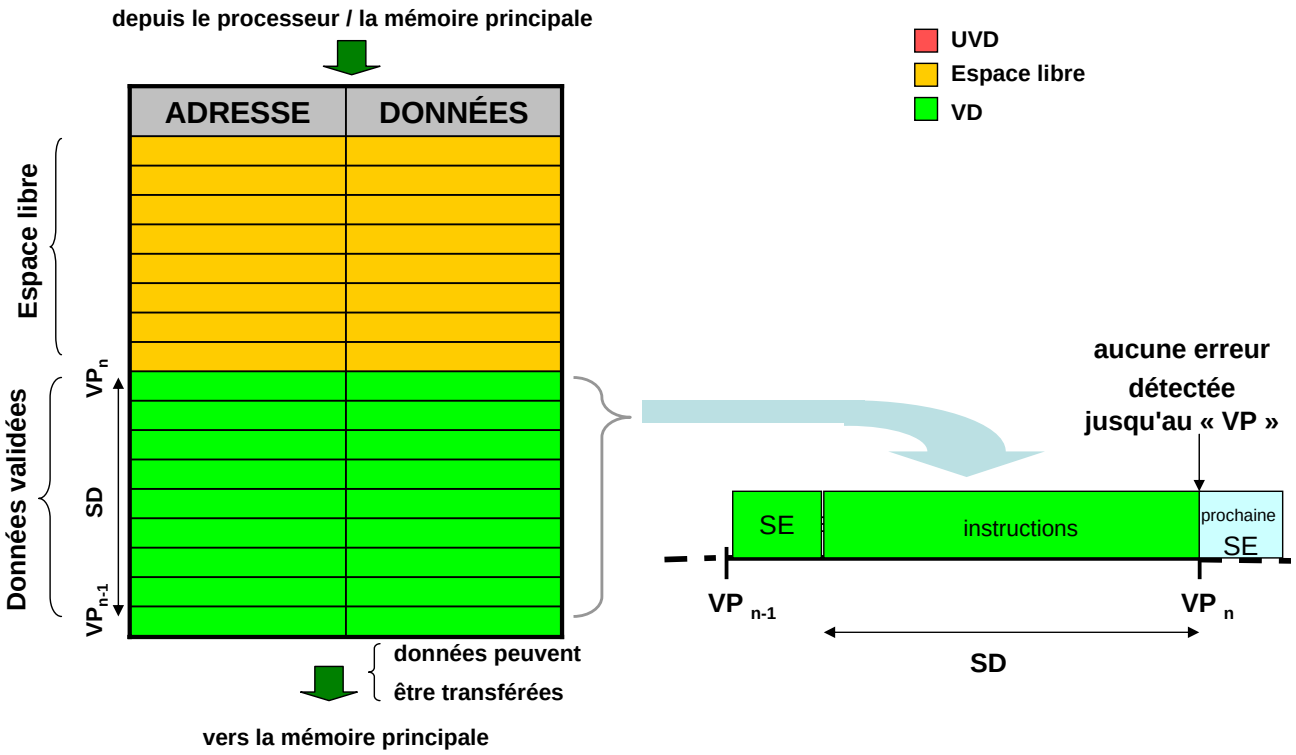


FIGURE 3.22 – Aucune erreur détectée lors de la séquence : les données sont validées au VP

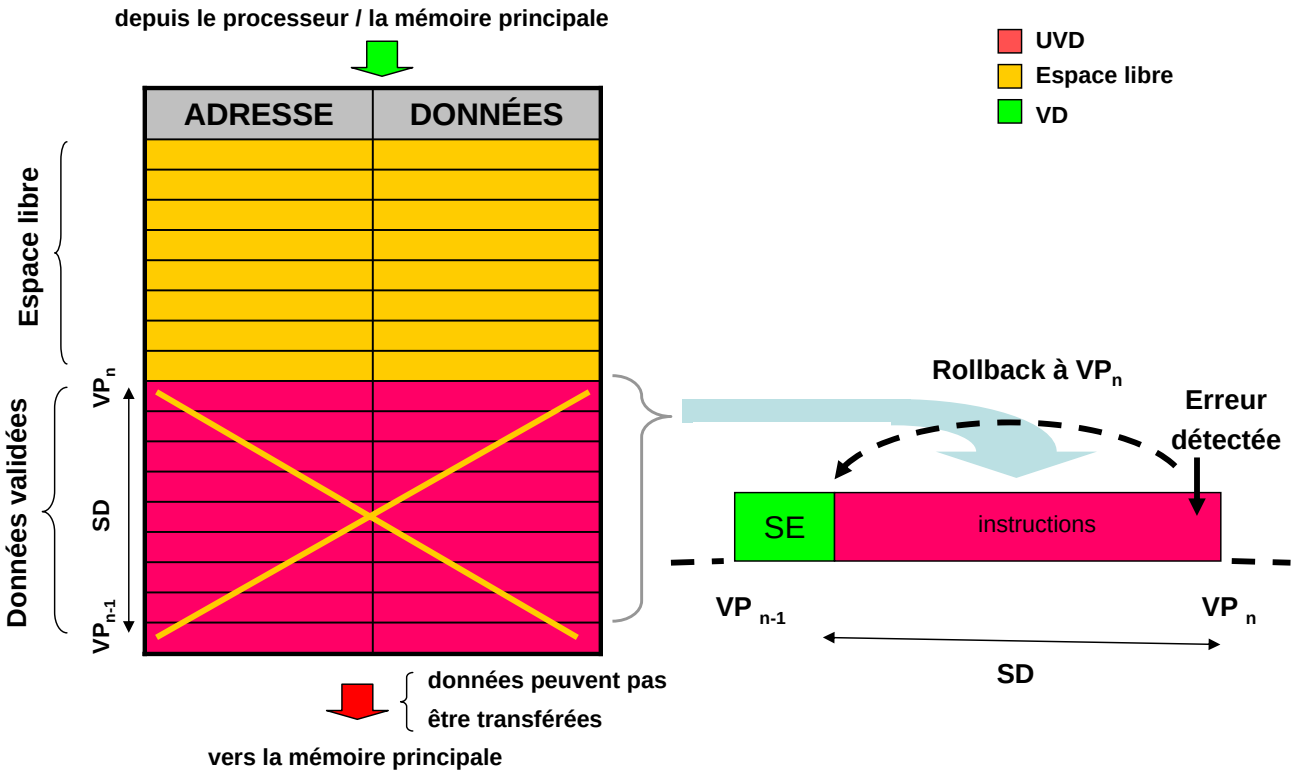


FIGURE 3.23 – Erreur détectée : toutes les données écrites au cours de la séquence sont supprimées

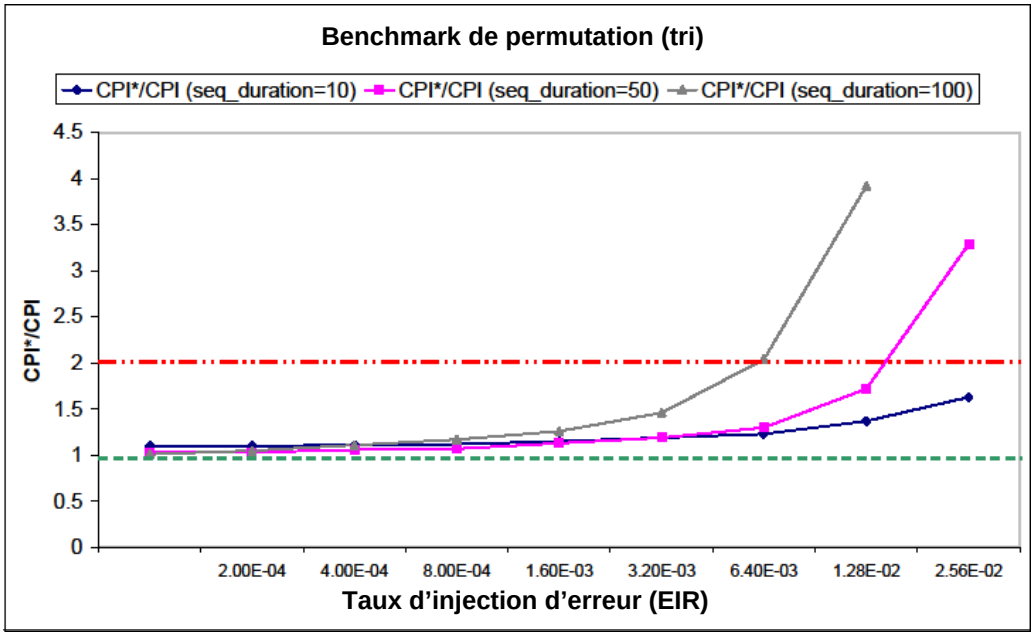


FIGURE 3.24 – Modèle II : CPI additionnels pour les benchmarks du groupe I

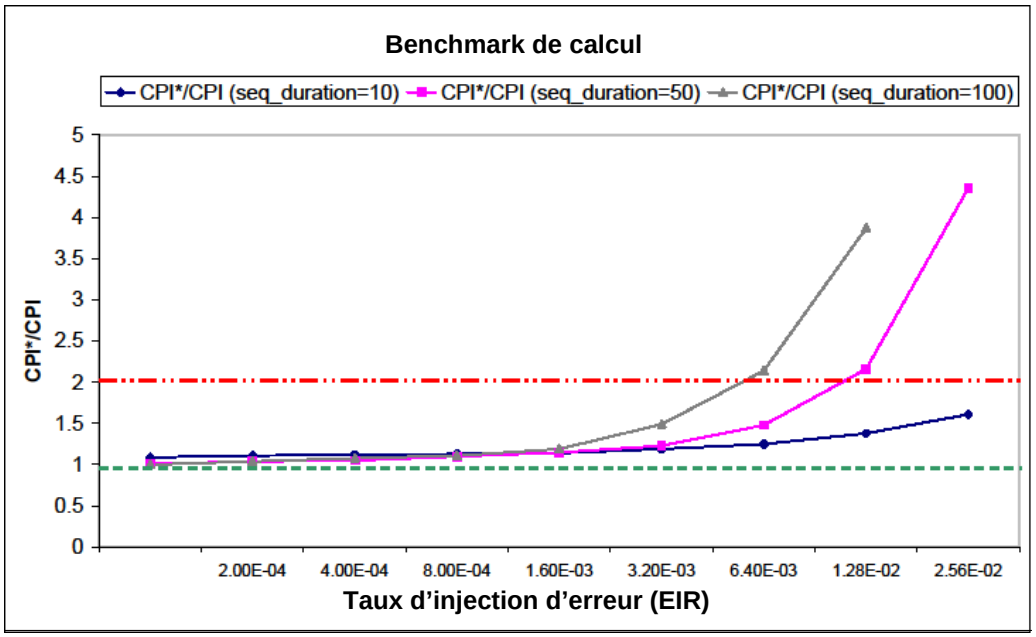


FIGURE 3.25 – Modèle II : CPI additionnels pour les benchmarks du groupe II

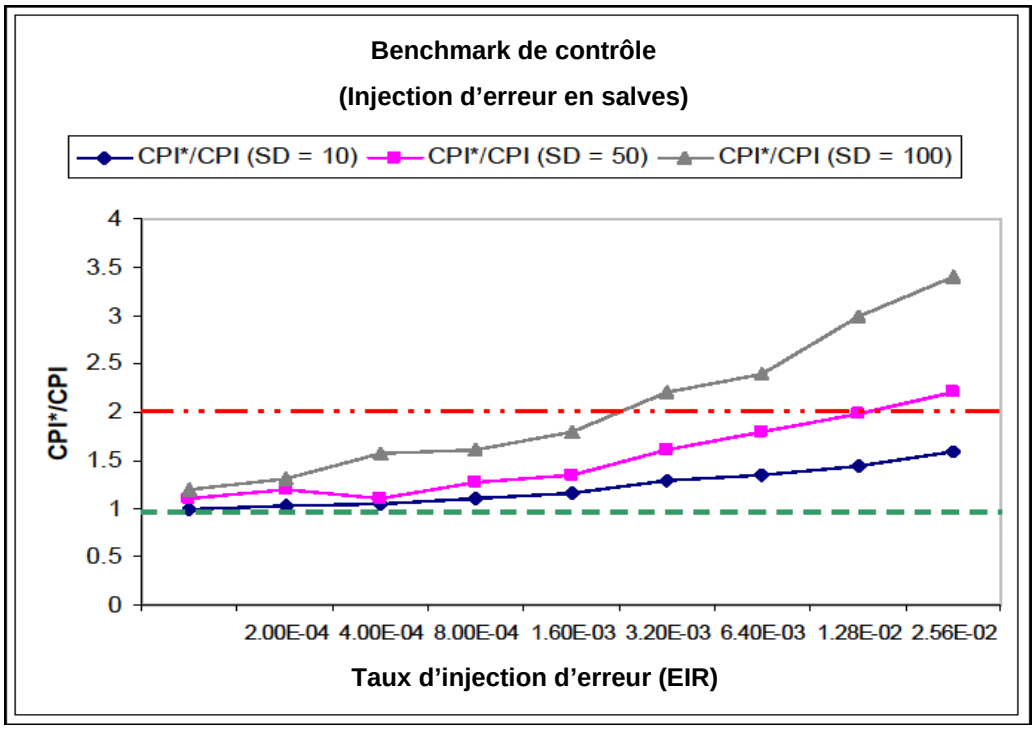


FIGURE 3.26 – Modèle II : CPI additionnels pour les benchmarks du groupe III

Chapitre 4

Conception et réalisation d'un processeur autocontrôlé

Nous avons pour objectif de concevoir un processeur tolérant aux fautes comportant deux blocs principaux : un cœur autocontrôlé *(Self Checking Processor Core*, SCPC) et un journal matériel autocontrôlé (*Self Checking Hardware Journal*, SCHJ). Dans ce chapitre, nous nous pencherons uniquement sur la conception du SCPC comme indiqué dans la figure 4.1.

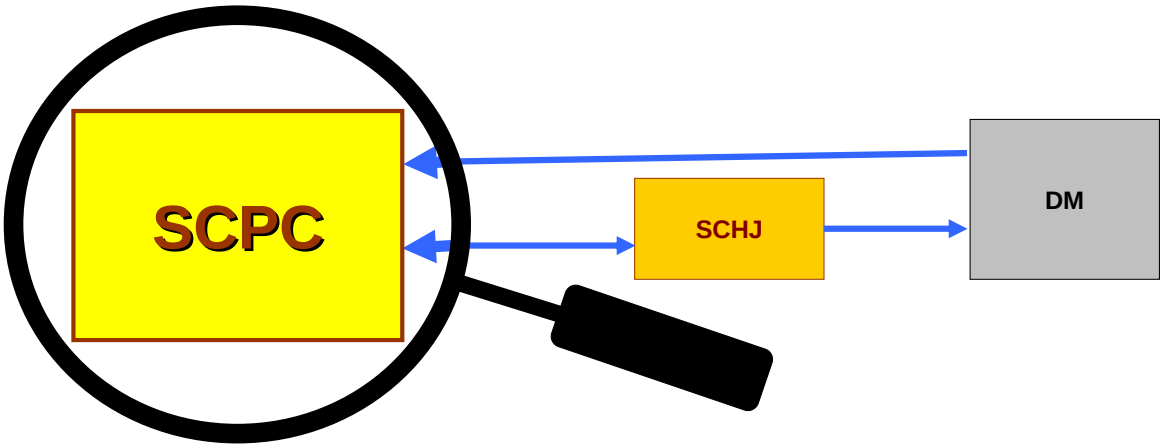


FIGURE 4.1 – Conception du cœur de processeur autocontrôlé

Pour décrire le SCPC, le chapitre a été divisé en plusieurs sections. Dans la première section, nous commençons la modélisation du processeur en choisissant une classe d'architecture appropriée qui réponde aux objectifs de conception identifiés dans le chapitre 3. Dans la section suivante, le modèle matériel du processeur – dans sa version non tolérante aux fautes – sera présenté et exploré. Les défis de performance et de fiabilité seront identifiés dans le modèle matériel. Les sections suivantes traitent de leurs solutions. Un modèle générique sera décrit en VHDL au niveau RTL (*Register Transfer Level*) et synthétisé à l'aide de l'outil Quartus II d'Altera. Les résultats expérimentaux seront présentés en termes de débit (nombre de bits traités par seconde) et d'utilisation de surface. Enfin, la capacité de tolérance aux fautes du SCPC sera validée dans le chapitre 6.

4.1 Stratégie de conception du processeur

La stratégie de tolérance aux fautes que nous avons choisi de mettre en œuvre a déjà été discutée dans la section 3.1. Dans cette partie, nous décrivons le choix d'une architecture de processeur qui réponde aux besoins, tel que présenté dans la figure 4.2 (suite de la figure 3.1 présentée précédemment). La détection d'erreur matérielle concurrente est chère si de nombreux états internes doivent être vérifiés, ce qui est contraire aux contraintes de conception énoncées. Ainsi, pour permettre une détection d'erreurs rapide et à faible surcoût matériel, il est nécessaire d'utiliser un processeur avec un minimum d'états internes à vérifier (cf figure 4.2 présentant les critères de conception du processeur). Le mécanisme de *rollback* logiciel implique un coût matériel faible, mais il sera lent si beaucoup d'états internes du processeur doivent être enregistrés périodiquement. De plus, le *rollback* en cas de détection d'erreur sera plus rapide si le nombre d'états internes à restaurer est plus faible.

Par conséquent, notre choix ira vers une architecture de processeur appartenant à la classe MISC (*Minimum Instruction Set Computer*), qui possède nombre des caractéristiques souhaitées et en fait une solution adaptée à notre stratégie de conception. Parmi les architectures de la classe MISC, nous avons choisi un processeur à pile de données. Il s'agit d'une architecture simple et flexible ayant un nombre très réduit de registres internes [JDMD07]. Un autre choix possible serait un processeur à accumulateur, mais son principal inconvénient est d'être très dépendant des accès à la mémoire et, par conséquent, d'être moins efficace. Cependant, l'architecture du processeur à pile choisi repose également sur de nombreux accès mémoire, mais la plupart d'entre eux sont très prévisibles (adresses voisines) car ils sont liés aux opérations sur la pile, et peuvent être gérés très efficacement.

4.1.1 Avantages du processeur à pile

Les processeurs à pile présentent de nombreux avantages, tant du point de vue de la protection que de la performance. Certains de ces avantages sont abordés dans [KJ89], et d'autres sont détaillés ci-dessous.

Les processeurs à pile présentent une architecture plus fiable comparée à celle de leurs homologues RISC, car ils possèdent moins d'états internes et présentent une faible surface sur la puce, ce qui réduit la probabilité d'une contamination par l'environnement externe. Dans la plupart des approches basées sur le paradigme RISC, on trouve une banque de registres qui les rend plus sensibles envers les SEU et MBU, alors que dans les processeur à pile le nombre de registres internes est nettement moindre. Par exemple, le processeur à pile de données présenté dans [Jal09] possède six registres internes : Le sommet de la pile de données (*Top of Stack*, TOS), le sous-sommet de la pile (*Next of Stack*, NOS), le sommet de la pile de retour (*Top of Return Stack*, TORS), le pointeur d'instruction (*Instruction Pointer*, IP), le pointeur de la pile de données (*Data Stack Pointer*, DSP) et le pointeur de la pile de retour (*Return Stack Pointer*, RSP). Ils sont donc beaucoup moins nombreux que dans les processeurs RISC modernes : à titre d'exemple, le processeur LEON 3 FT possède plus de 150 registres internes [Aer11]). Or dans les processeurs tolérants aux fautes, tous les registres internes doivent être protégés contre les fautes transitoires [ARM+11].

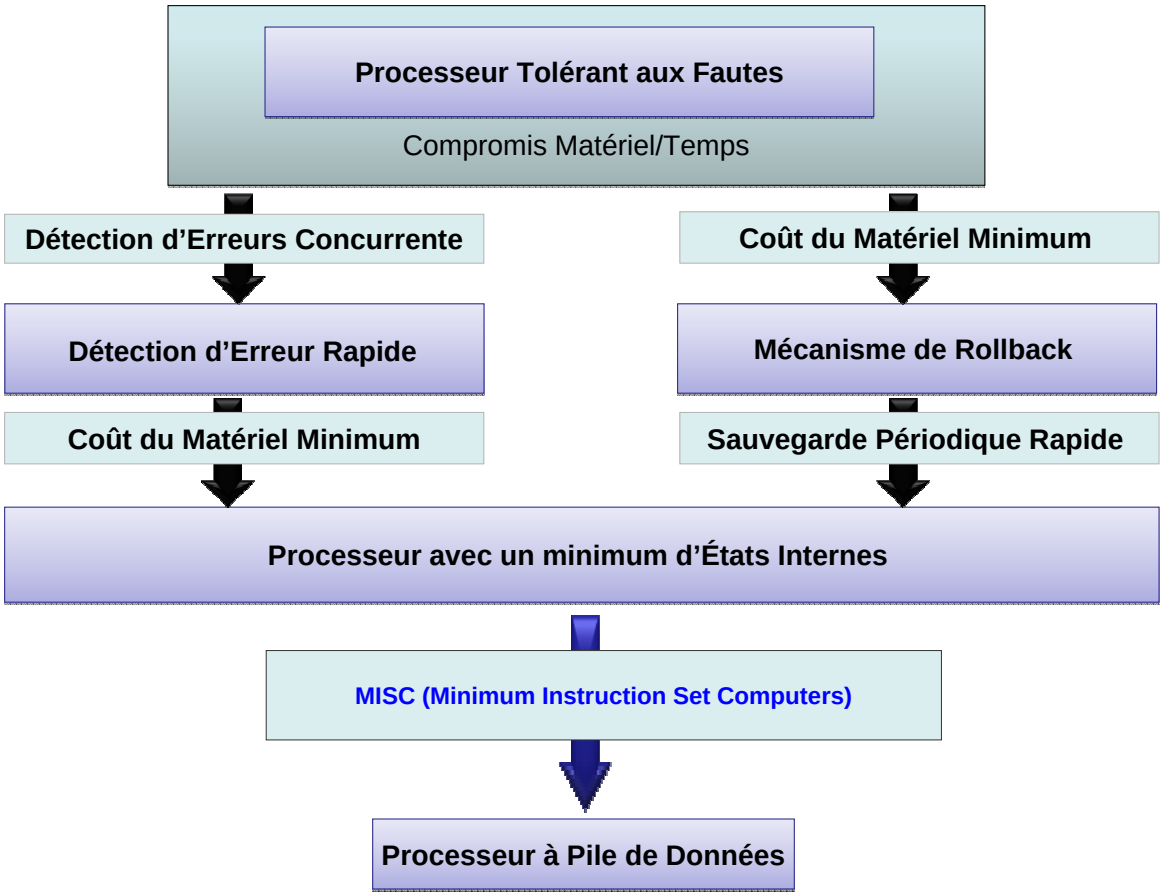


FIGURE 4.2 – Critères du choix du processeur à pile de données

Par ailleurs, en raison du nombre élevé de registres internes dans les architectures RISC, la longueur des instructions est importante. Plus de registres implique également un décodage d'adresse plus évolué, ce qui augmente le délai de propagation. C'est pourquoi, les processeurs RISC (et CISC) modernes ont besoin de plusieurs étages de pipeline pour rétablir un débit moyen correct en masquant la latence interne. Par ailleurs, ils nécessitent un meilleur ordonnancement des branchements. Par exemple, le Pentium 4 possède un pipeline de 20 étages, et un échec lors des accès aux caches ou aux buffers de prédiction de branchement peut impliquer une pénalité de 30 cycles en cas de mauvaise prédiction de branchement (20 cycles perdus dans le pipeline, 10 dans les accès à la mémoire). À l'opposé, le RTX (modèle de processeur à pile) a un dépassement fixe de 2 cycles dans tous les cas [PB04]. Par ailleurs, la résistance naturelle d'un processeur contre les SEU diminue avec l'augmentation du nombre d'étages du pipeline [MW04].

Les processeurs à pile ont d'autres avantages par rapport aux RISC, tels que des fréquences d'horloge plus élevées, une pénalité moindre lors des appels de procédures, et une gestion des interruptions plus rapide [Sha06]. Ils possède une fréquence d'horloge plus élevée parce que les instructions sont exécutées entre les deux sommets de la pile, à condition qu'il y ait une mise en cache interne de la pile, ou une pile matérielle interne. La perte de temps lors des appels de procédures est réduite car il y a un nombre limité de registres à sauvegarder en mémoire lors du branchement à l'adresse de

la procédure appelée. Enfin, une gestion plus rapide des interruptions est possible car les routines d'interruption peuvent être exécutées immédiatement de par le fait que le matériel prend en charge la gestion de la pile. Une architecture de processeur Java basée sur une pile a été évaluée dans [Sch08], et les résultats montrent une meilleure performance et un nombre plus faible de portes par rapport à un processeur de la famille RISC implanté sur FPGA.

Au niveau commercial, des processeurs à pile ont été utilisés pour l'imagerie médicale, dans des disques durs, ainsi que dans des applications satellitaires. Parmi les processeurs utilisés dans l'industrie, on peut citer les Novix NC4000, Harris RTX2000, ou Silicon Composers SC32 [PB04]. Ils sont déployés dans des applications spatiales pour leur performance raisonnable et leur faible consommation [HH06], comme par exemple le SCIP, un processeur à pile conçu pour les véhicules spatiaux [Hay05]. Plus récent, le projet Green-Arrays utilise des processeurs à pile pour la conception de puces multiprocesseurs. L'entreprise a conçu des puces avec des caractéristiques intéressantes, comme un coût de fabrication faible et une consommation minimale avec des performances élevées [Gre10, Bai10].

La conception du processeur tolérant aux fautes tient compte de notre objectif à long terme : l'élaboration d'un système multi-ressources tolérant aux fautes basé sur le passage de messages, dans lequel ce processeur tolérant aux fautes sera utilisé comme nœud de traitement. Il est clair dès le départ que des contraintes sévères concernant la consommation de surface s'appliquent à la conception architecturale d'un seul nœud afin de permettre à l'avenir de répondre aux objectifs de conception massivement parallèle, tout en préservant le plus possible les performances individuelles de chacun des nœuds. Les processeurs à pile restent une architecture viable, en raison des besoins en circuits de petite taille, à coût modéré, et faible consommation électrique. Les processeurs à pile autorisent la réalisation de cœurs simples pour les applications parallèles distribuées [Gre10].

D'autre part, les processeurs à pile sont des architectures adaptées à l'exécution d'instructions séquentielles. Ils correspondent bien aux besoins des applications où le contrôle est dominant. Ils sont cependant moins adaptés aux applications où le traitement de données est dominant, comme le streaming vidéo.

4.2 Architecture proposée

L'architecture du processeur à pile a été présenté dans [Jal09]. Elle est inspirée de la seconde génération de processeur à pile canonique [KJ89]. La taxonomie de la pile est basée sur trois attributs : le nombre de piles, la taille des piles, et le nombre d'opérandes dans les instructions. Ils sont représentés dans la figure 4.3 par trois axes de coordonnées. Ces dimensions ont plusieurs combinaisons possibles. Parmi ces choix, le processeur à pile canonique est à piles multiples, de grande taille, et ses instructions sont à zéro opérande (ML0), comme indiqué dans la figure 4.3. Le terme « zéro opérande » signifie que l'emplacement des opérandes de toutes les instructions est implicite, et qu'il n'est donc pas nécessaire de préciser leur adresse dans l'instruction. Dans le cas des processeurs à pile de donnée, l'emplacement implicite est le sommet de la pile.

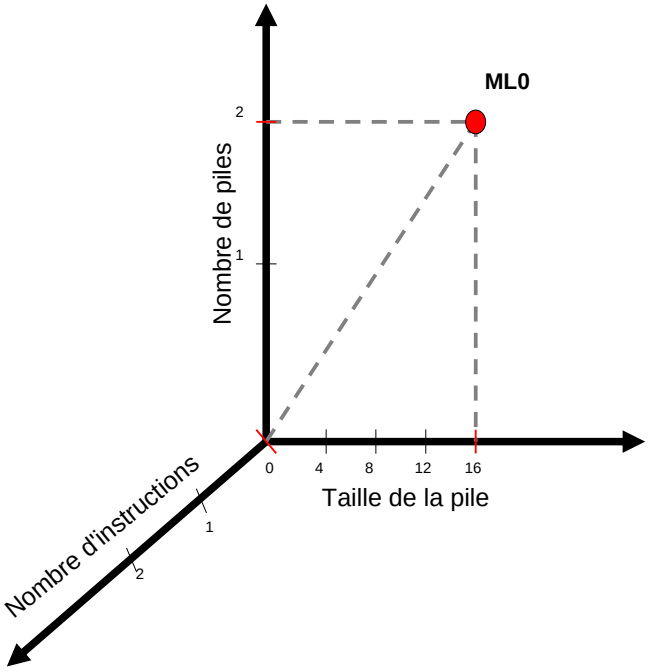


FIGURE 4.3 – Processeur à piles multiples, de grande taille, et zéro opérande (ML0)

Pour satisfaire à l'exigence de simplicité, il possède deux piles [1], une pile de données (*Data Stack*, DS), et une pile de retour (*Return Stack*, RS). La première est utilisée pour l'évaluation des expressions – traitement des données – et le passage des paramètres aux sous-programmes, et la seconde pour l'adresse de retour des sous-programme, l'adresse des routines d'interruption et les copies temporaires des données. L'existence de deux piles permet d'accéder aux différentes valeurs en un seul cycle d'horloge et améliore la vitesse de traitement. Du fait des deux piles séparées pour les adresses de retour et les données, les appels de routines peuvent être effectués en parallèle avec les opérations sur les données. Cela permet de réduire la taille du programme et la complexité du système, ce qui améliore également les performances.

Concernant les buffers de pile, nous avons choisi de leur attribuer une grande taille et de les faire résider dans la mémoire sûre principale, ce qui permet le stockage de nombreuses données sans perte. La mémoire est sur la même puce que le cœur, afin que les données soient accessibles en un seul cycle d'horloge. De plus, le fait d'utiliser la mémoire principale pour la pile permet de n'avoir aucune restriction sur la profondeur de celle-ci.

Comme vu précédemment, le processeur comporte trois registres nommés TOS, NOS, et TORS, qui représentent les sommet et sous-sommet de la pile de données (DS), et le sommet de la pile de retour (TORS), respectivement. NOS et TORS n'existent pas dans le modèle canonique. Leur utilité a été prouvée dans [Jal09], car ils permettent de simplifier le jeu d'instructions. Les piles DS et RS résident dans la mémoire principale, ce qui est une caractéristique similaire à la première génération de processeurs à pile. Elles n'ont pas de registre d'adresse, mais sont adressées par des pointeurs internes, à savoir : le pointeur de la pile de données (*Data Stack Pointer*, DSP) et le pointeur de la

[1] Selon la définition de Turing, le nombre minimal de piles pour une machine à pile pure est de 2 [KJ89]

Tableau: Liste des acronymes

DS	**D**ata **S**tack
RS	**R**eturn **S**tack
I/O	**I**nput-**O**utput
NOS	**N**ext **O**f **S**tack
TORS	**T**op **O**f **R**eturn **S**tack
TOS	**T**op **O**f **S**tack
ALU	**A**rithmetic **L**ogic **U**nit
IP	**I**nstruction **P**ointer

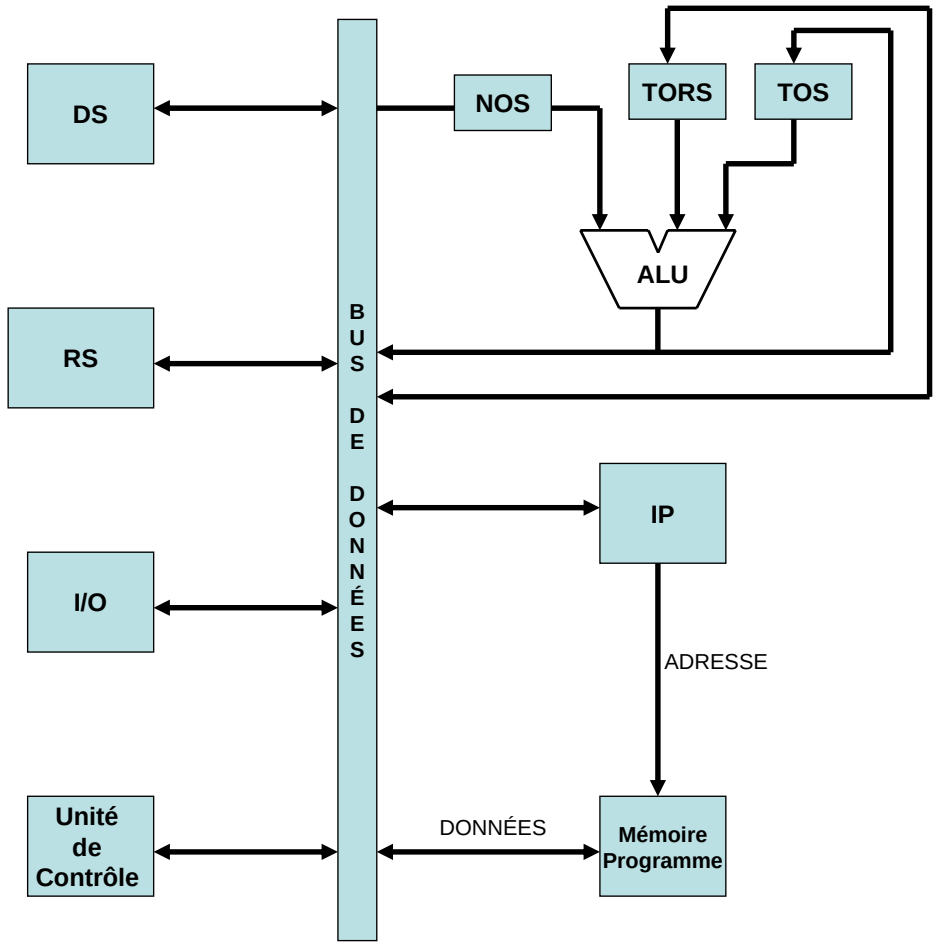


FIGURE 4.4 – Processeur à pile simplifié

pile de retour (*Return Stack Pointer*, RSP). Nous avons choisi ces caractéristiques afin de protéger le contenu des données car selon l'hypothèse faite en préambule de ce travail, la mémoire principale (*Dependable Memory*, DM) étant une zone de stockage sûre, ces piles restent ainsi protégées contre les fautes.

L'architecture proposée pour le processeur à pile contient un bus de données, la pile de données (DS) et de retour (RS) avec leurs registres des sommets de piles, l'unité arithmétique et logique (ALU), le registre pointeur d'instruction (IP), le buffer d'instruction avec le registre d'instruction et sa logique de contrôle, comme illustré par la figure 4.4. Le module d'entrée-sortie nécessite une gestion particulière pour être tolérant aux fautes et ce point n'est pas abordé dans ce travail.

L'ALU effectue les opérations arithmétiques et logiques, ce qui comprend l'addition, la soustraction, les fonctions logiques (AND, OR, XOR), le test de zéro et d'autres. Elle effectue les opérations sur le sommet et le sous-sommet de la pile de données (opérandes et résultat), TOS et NOS étant les deux premiers éléments de DS. Le registre IP contient l'adresse de la prochaine instruction à exécuter. Il peut être chargé à partir du bus pour mettre en œuvre les branchements, ou peut être incrémenté pour aller chercher l'instruction suivante en mémoire programme (*Program Memory*, PM) lors d'un fonctionnement séquentiel. Comme les piles de données et de retour, la mémoire programme réside

également en mémoire principale.

Le MAR (*Memory Address Register*), unité qui existe dans le processeur à pile canonique, a été éliminé de notre modèle, car la mémoire programme associée avec le registre IP est suffisante pour gérer toutes les instructions et générer l'adresse de la prochaine instruction à exécuter. Le processeur proposé a un jeu d'instructions simple de 37 instructions qui peuvent toutes être exécutées en un seul cycle d'horloge à l'exception de l'instruction STORE qui en nécessite deux. Dans la version pipelinée, l'ensemble du jeu de 37 instructions est présenté en annexe B où toutes les instructions sont exprimées au niveau RTL (*Register Transfer Level*). Du fait d'un jeu d'instructions limité et du modèle à zéro opérande, un opcode de 8 bits est suffisant pour représenter toutes les instructions.

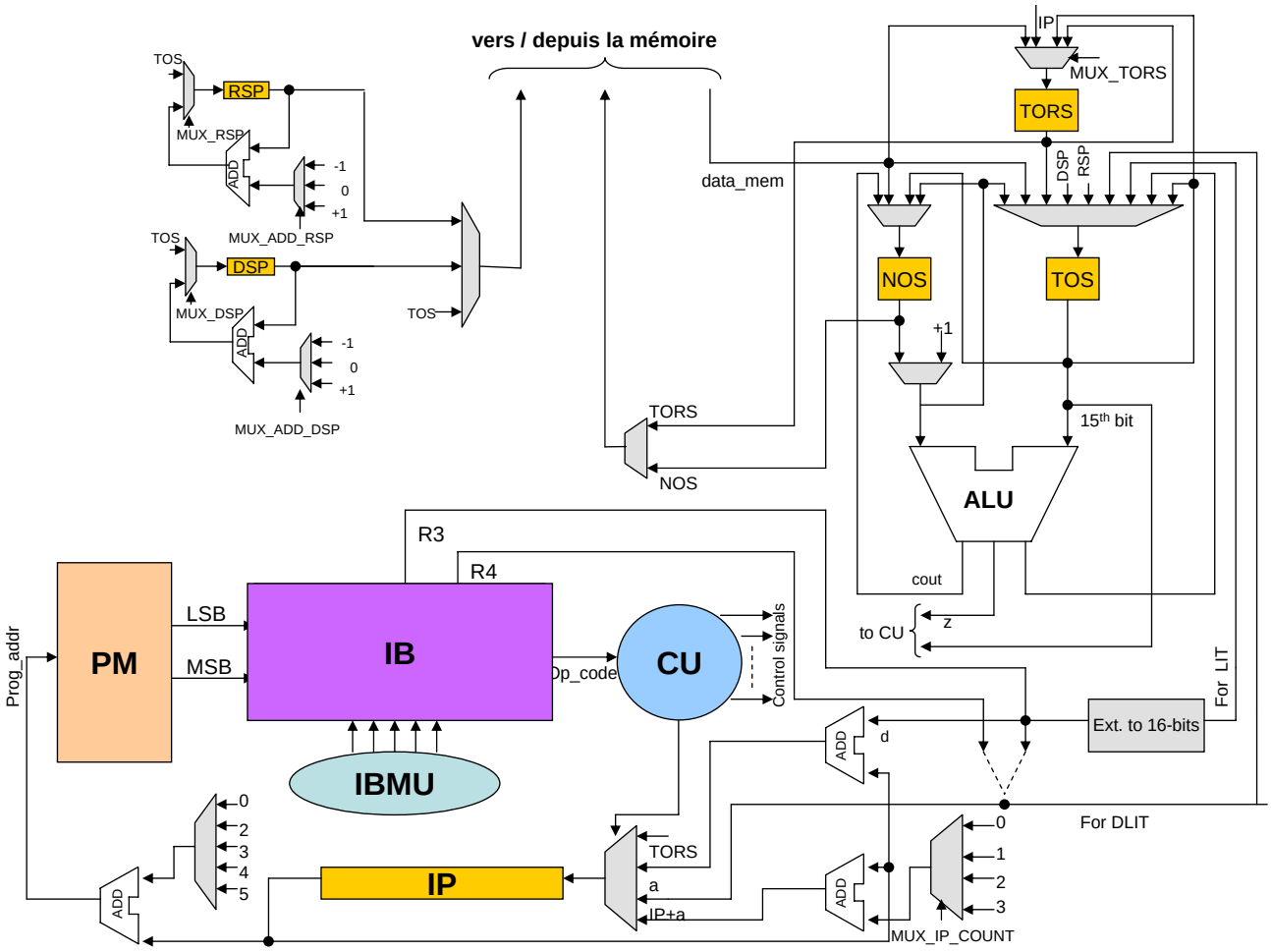


FIGURE 4.5 – Modèle de processeur à pile

4.3 Modèle matériel du processeur à pile

Le modèle matériel du processeur a été décrit en VHDL au niveau RTL. Le modèle de processeur initial (dans sa version non tolérante aux fautes) a été synthétisé avec Altera Quartus II.

Il se compose de l'unité arithmétique et logique (ALU), des registres internes, du buffer d'instructions, de l'unité de contrôle et du chemin de données, comme le montre la figure 4.5. Les piles DS et RS sont adressées par les deux registres pointeurs DSP et RSP respectivement. Les trois registres intégrés TOS, NOS et TORS résolvent les possibles conflits lors du transfert de données entre les deux piles, par exemple pendant l'exécution des instructions `R2D`, `D2R`, `OVER`, ou `ROT`. À titre d'exemple, considérons l'instruction `R2D`(*Return Stack to Data Stack*). En raison de la disponibilité des TORS, TOS et NOS dans le cœur du processeur, aucun conflit d'accès au bus de données ne se produit : le contenu de TORS est copié dans TOS ; TOS est copié dans NOS ; le DSP est incrémenté ; NOS est rappatrié vers DS ; le contenu de RS [RSP] est chargé dans TORS ; finalement, RSP est décrémenté. Par conséquent, aucun conflit ne survient.

Dans un processeur à pile, l'exécution des instructions est normalement plus rapide que dans les processeurs classiques, car les données sont implicitement disponibles sur les deux sommets de la pile, plutôt que d'avoir à lire les données dans des registres ou à une adresse mémoire. La taille du chemin critique est donc réduite de façon notable. Pour une meilleure compréhension, le chemin de données simplifié pour les instructions arithmétiques et logiques est décrit dans la figure 4.6. Le processeur lit la mémoire en parallèle afin de compenser l'effet du déséquilibre dû à « un élément en moins » dans la pile. La lecture de la mémoire a uniquement pour but de remplir l'espace laissé vide par l'exécution de l'instruction (pour préparer l'instruction suivante). Par conséquent, il n'a pas besoin d'attendre le décodage d'adresse avant d'accéder à des opérandes.

Généralement, chaque bloc de la mémoire programme (16 bits) contient deux instructions successives (8 bits + 8 bits). Les instructions qui résident dans la mémoire programme traversent le buffer d'instructions (IB) et sont décodées dans l'unité de contrôle, qui active tous les multiplexeurs en conséquence. L'IB est alimenté par une paire de LSB suivie par les MSB, comme indiqué dans la figure 4.18. L'IB est constitué de buffers 8 bits en cascade, reliés via des multiplexeurs qui contrôlent le flot d'instructions (comme illustré par la figure 4.18). L'interconnexion entre les multiplexeurs est contrôlée par l'unité de gestion du buffer d'instruction (*Instruction Buffer Management Unit*, IBMU), comme le montre la figure 4.6). L'IB et l'IBMU seront présentés plus tard en détail.

Bien que le processeur à pile ait un opcode de longueur fixe, quelques instructions ont besoin d'information supplémentaire (donnée immédiate) pour être exécutées. Ainsi, le bloc d'instruction est parfois supérieur à un octet. Ces instructions comprennent les branchements et le `CALL` qui nécessitent 16 bits supplémentaires pour l'adresse (ou 8 bits supplémentaires de déplacement) pour connaître l'adresse cible (par exemple, voir l'annexe C pour `ZBRA d`, `SBRA d`), ainsi que les instructions nécessitant une donnée immédiate constante (par exemple, `LIT a`, `DLIT a`). Ce défi est relevé dans la section 4.4.2.

L'unité de contrôle gère les composants du processeur ; elle lit et décode les instructions, les transforme en une série de signaux de contrôle qui activent d'autres parties du processeur via des multiplexeurs. Les rôles principaux de l'unité de contrôle sont :

- décoder le code numérique de l'instruction pour générer un ensemble de signaux pour chacun des MUX ;

- mettre à jour les pointeurs DSP et RSP ;
- activer la lecture ou l'écriture en mémoire en fonction de l'instruction active ;
- sélectionner la bonne opération de l'ALU.

Le pointeur d'instruction IP pointe vers la prochaine instruction à exécuter. Il prépare la mémoire programme de façon à alimenter le buffer d'instructions en fonction des besoins pour l'exécution de cette instruction.

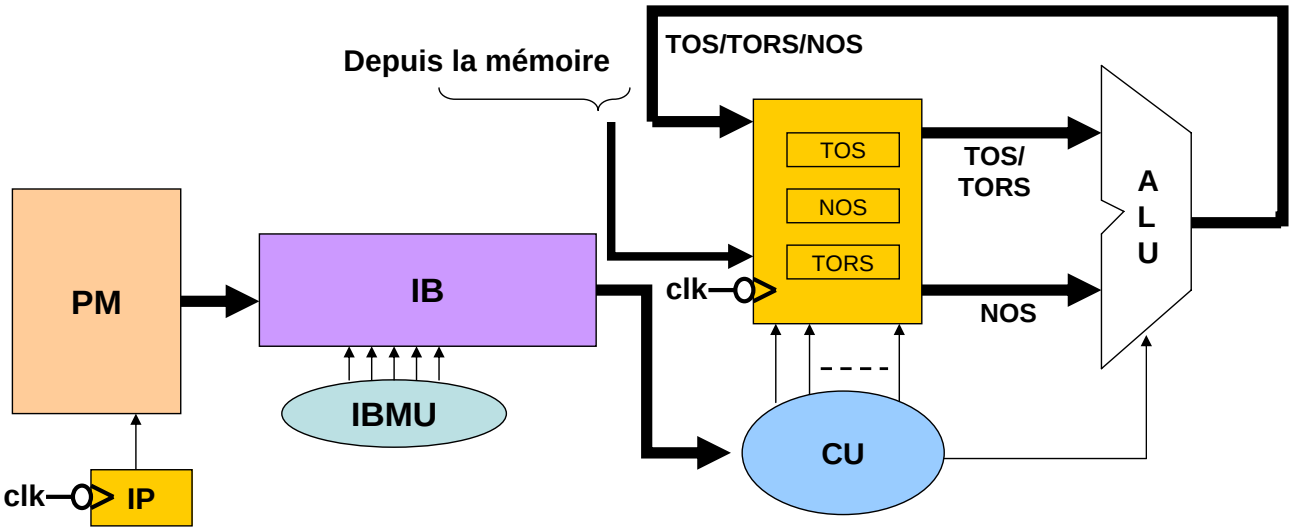


FIGURE 4.6 – Chemin de données simplifié du modèle proposé (instructions arithmétiques et logiques)

L'exécution de branchements conditionnels et inconditionnels a été discutée dans [KJ89] et explorée plus profondément pour la conception du processeur à pile modifié [Jal09]. Le processeur à pile est rapide dans l'exécution des branchements en raison d'un nombre minimal d'étages de pipeline [PB04]. Cependant, chaque instruction de branchement est suivie par une instruction `NOP` (*no-operation*), car l'IB est vidé afin de charger de nouvelles instructions, ce qui implique une pénalité en temps. Cette question n'a pas été abordée dans ce travail, cependant une solution possible a été proposée dans la section 4.6.3.

4.4 Les défis de conception du processeur à pile tolérant aux fautes

Cette section est dédiée à l'implantation du processeur à pile sur la base de la méthodologie de conception tolérante aux fautes. L'architecture requise doit avoir une capacité d'autocontrôle avec de bonnes performances. Ces deux défis sont abordés dans cette section.

4.4.1 Défi n°I : mécanisme d'autocontrôle

Le fait que l'architecture ait un nombre minimum de registres internes ne garantit pas qu'il n'y ait aucune possibilité d'apparition d'erreurs. Les perturbations externes peuvent toujours compromette

le fonctionnement correct du processeur, même si les probabilités sont plus faible qu'avec les autres classes de processeurs, RISC ou CISC. Il est donc nécessaire que les registres internes et l'ALU puissent s'autocontrôler.

4.4.2 Défi n°II : amélioration des performances

Dépendant des choix architecturaux effectués pour mettre en œuvre processeur à pile, deux facteurs impactent les performances en limitant la vitesse d'exécution : (i) l'exécution d'instructions multi-horloges et (ii) les blocs d'instructions multi-octets. Ces deux facteurs ajoutent des délais supplémentaires dans l'exécution des programmes.

Défi n°II.a : exécution des instructions multi-horloges

La plupart des instructions nécessitent un seul cycle d'horloge pour être exécutées, mais quelques instructions en nécessitent plusieurs, comme `DUP`, `OVER`, `R2D`, `CPR2D`, `D2R`, `FETCH`, `STORE`, `PUSH_DSP`, `PUSH_RSP`, `LIT`, `DLIT`, `CALL`. Leur nombre minimal de cycles ne peut pas être réduit à un avec une architecture sans pipeline en raison des conflits d'accès aux bus de données dans la même direction.

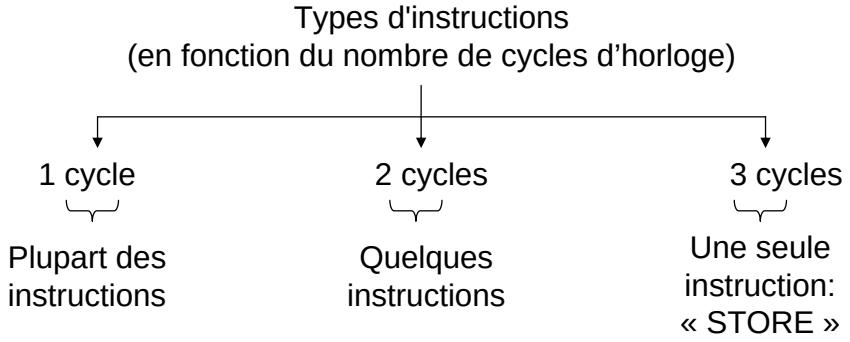


FIGURE 4.7 – Différents types d'instructions du point de vue de leur exécution (sans pipelining)

Pour une meilleure compréhension, examinons l'instruction `DUP` (duplication) qui nécessite deux cycles d'horloge pour sa complète exécution. Ici, le contenu de TOS doit être copié dans NOS et le contenu de NOS transféré à la troisième position de la pile de données DS, pointée par le DSP. Si l'instruction était exécutée en un cycle d'horloge, alors nous perdrions les données à la troisième position de la pile. En effet, le contenu de NOS étant écrit à l'adresse pointée par le DSP, sans incrémentation préalable du DSP la donnée à l'adresse pointée serait écrasée.

Cependant, elle peut être exécutée avec succès en deux cycles d'horloge. Au premier cycle (t) un nouvel emplacement est créé en incrémentant DSP, et au cycle suivant ($t + 1$) le contenu de TOS est copié dans NOS et le contenu de NOS sauvegardé dans la pile à l'emplacement pointé (DS[DSP]), comme le montre la figure 4.8. De telles instructions multi-cycles entraînent des dégradations de performance et nécessitent la mise en œuvre d'un pipeline.

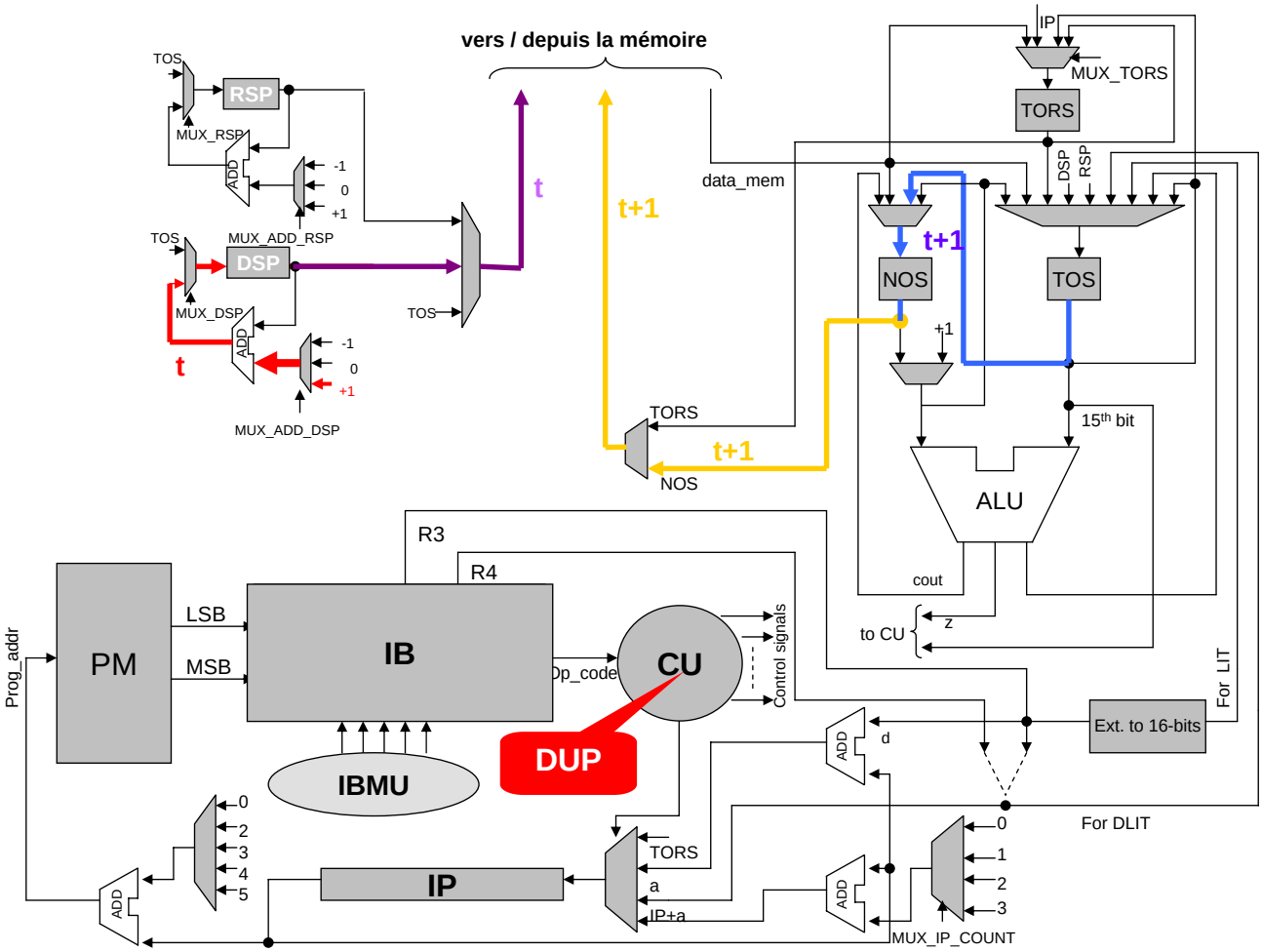


FIGURE 4.8 – Exécution de l'instruction duplication (DUP) en 2 cycles d'horloge

Défi n°II.b : instructions multi-octets

L'opcode des instructions a une longueur d'un octet. Il utilise des registres source et destination implicites, et ne nécessite donc pas d'adressage explicite. Par exemple, l'instruction `ADD` (addition) signifie ajouter TOS avec NOS, et stocker le résultat dans TOS. Toutefois, 7 des 37 instructions nécessitent un paramètre supplémentaire à fournir : une valeur constante immédiate de 8 ou 16 bits (`LIT` et `DLIT` respectivement), une adresse absolue de 16 bits (`LBRA`, `CALL`), ou un déplacement de 8 bits (`SBRA`, `ZBRA`) (voir annexe C, tableau 8).

D'autre part, la mémoire programme est de 16 bits alors l'opcode est de 8 bits. Le débit moyen des instructions en cours d'exécution est donc inférieur à la capacité de pré-chargement, la première étant proche de 8 bits par cycle, tandis que la seconde est plus proche de 16 bits par cycle. Par conséquent, le chargement des instructions dans le buffer d'instructions IB est presque deux fois supérieur au taux d'exécution (en considérant qu'un opcode de 8 bits est exécuté en un cycle d'horloge). Cela nécessite une gestion intelligente du buffer d'instruction pour (i) contrôler le flot d'entrée et de sortie des instructions, et (ii) gérer le flot d'instructions à taille de bloc variable (telles que `LIT`, `LBRA`).

Pour exécuter une instruction en un seule cycle, la prochaine instruction à exécuter doit rejoindre

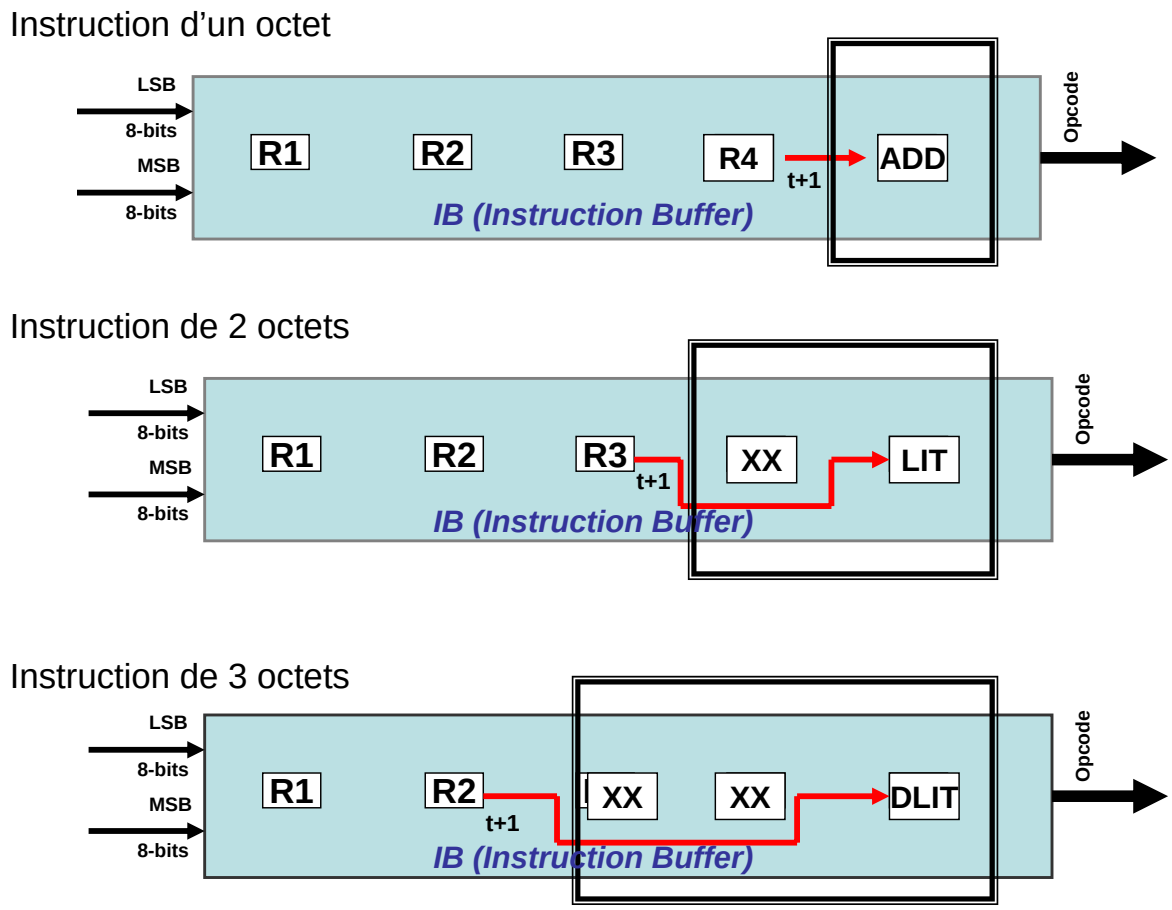


FIGURE 4.9 – Instructions multi-octets

l'unité de contrôle au cycle d'horloge suivant $(t+1)$. Par exemple, la figure 4.9 aborde cette question. Tout d'abord, nous supposons que l'instruction en cours d'exécution est `ADD` (l'instruction à exécuter réside dans le registre R5). La taille de cette instruction est de 1 octet, donc l'opcode de l'instruction suivante doit être en écrit dans R4. Cet opcode doit atteindre l'unité de contrôle dans le cycle d'horloge suivant $(t+1)$. Deuxièmement, nous supposons `LIT a` comme une instruction active (au temps t) ; ainsi au cycle d'horloge suivant $(t+1)$ le contenu de R3 devrait rejoindre l'unité de contrôle.

Les solutions aux défis mentionnés ci-dessus seront abordées dans les sections suivantes.

4.5 Solution n°I : mécanisme d'autocontrôle

Le processeur a besoin de détecter les erreurs dans ALU et dans les états internes. Tout d'abord, nous commencerons par la conception d'une ALU autocontrôlée.

4.5.1 Détection d'erreurs dans l'ALU

Il n'existe aucun code unique qui puisse simultanément protéger les opérations arithmétiques et logiques. Par conséquent, nous utilisons la combinaison de codes arithmétiques et logiques (souvent

appelés « codes de combinaison ») pour protéger les deux types d'opérations : un code à résidu modulo 3 pour protéger les opérations arithmétiques, et un code à parité pour protéger les opérations logiques. Nous avons choisi ces codes car ils sont simples à mettre en œuvre et suffisamment efficaces pour valider l'efficacité de notre approche. Par ailleurs, ils requièrent un minimum de ressources, ce qui peut être présenté à partir des résultats publiés dans [SFRB05]. Dans [SFRB05], l'ALU est conçue avec différentes techniques de détection d'erreurs et a été simulée à l'aide de l'outil de simulation Quartus II fourni par Altera. L'utilisation des ressources du FPGA pour les deux techniques de détection d'erreurs intégrées (*Built-in error detection*, BIED) – le code de Berger et les codes à résidu ou de parité – a été relevée à partir des simulations effectuées. La figure 4.11 montre le tableau comparatif de l'utilisation des ressources avec les deux techniques BIED par rapport aux besoins d'une ALU TMR et d'une ALU sans détection d'erreur.

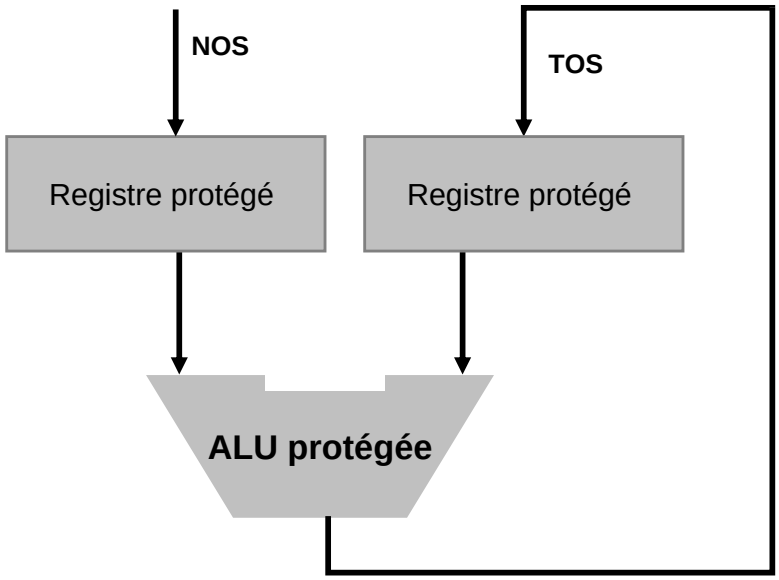


FIGURE 4.10 – Chemin de données protégé de l'ALU

Il est évident à partir de la figure 4.11 [SFRB05] que l'EDALU (ALU à détection d'erreur par résidu et contrôle de parité) utilise 54 % d'éléments logiques en moins que l'ALU TMR, et que l'ALU protégée par code de Berger en consomme 42 % en moins. Par conséquent, selon ces résultats, il est clair que l'ALU protégée par résidu et parité offre une meilleure utilisation des ressources que celle protégée par les codes Berger.

Les instructions de l'ALU appartiennent à deux groupes : arithmétique et logique (voir figure 4.12). En regroupant les instructions, la surface active du circuit à tout instant est réduite [SFRB05]. Par exemple, un SEU intervenant sur le module de génération de parité arithmétique n'affecterait pas le module logique et inversement.

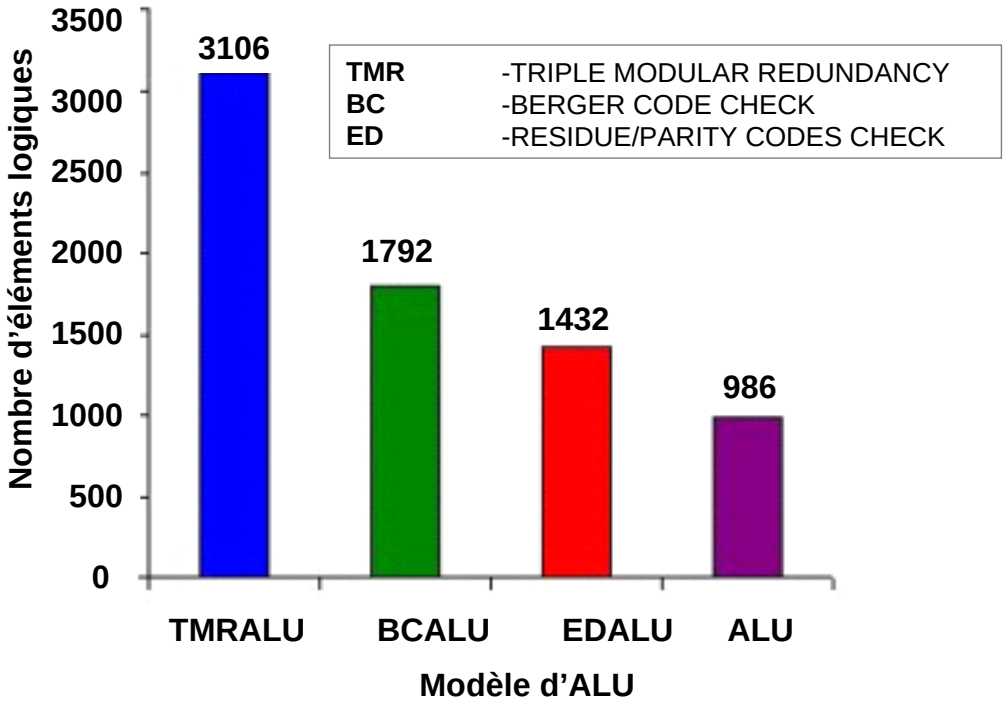


FIGURE 4.11 – Carte d'utilisation des ressources pour les différentes ALU [SFRB05]

Détection d'erreurs dans les instructions arithmétiques

Un reste calculé à partir des symboles de données X et Y peut préserver les opérations arithmétiques dans une ALU. Ici, la détection des erreurs dans les instructions arithmétiques est basée sur les résidus modulo 3. Dans l'ALU, ils sont calculés dans deux opérations concurrentes (comme montré dans la figure 4.13). D'une part, les deux opérandes X et Y se voient appliquer une opération arithmétique et les résultats sont stockés dans S. D'autre part, les résidus PA_X and PA_Y se voient appliquer l'opération arithmétique équivalente et génèrent le résidu. Le résultat PA_S sera stocké avec S (comme indiqué dans la figure 4.13). Au cycle d'horloge suivant, le générateur de parité modulo 3 va produire PA'_S (résidu de S), qui sera comparé à la parité déjà stockée PA_S. En cas de divergence, un signal d'erreur sera positionné.

Les mathématiques sur lesquelles les codes à résidu reposent sont présentées ci-dessous :

$X = x_n x(n-1)x(n-2).....x_2 x_1 x_0$,

$Y = y_n y(n-1)y(n-2).....y_2 y_1 y_0$,

où X et Y sont les données/symboles appliqués à l'entrée de l'ALU.

$C = C_m C(m-1)C(m-2).....C_2 C_1 C_0$

où C est le diviseur de contrôle utilisé pour calculer le résidu.

Les restes déterminés à partir de la division des données X and Y par le diviseur de contrôle C sont donnés par :

$PA_X = Rx_m Rx(m-1).....Rx_2 Rx_1 Rx_0$,

$PA_Y = Ry_m Ry(m-1).....Ry_2 Ry_1 Ry_0$

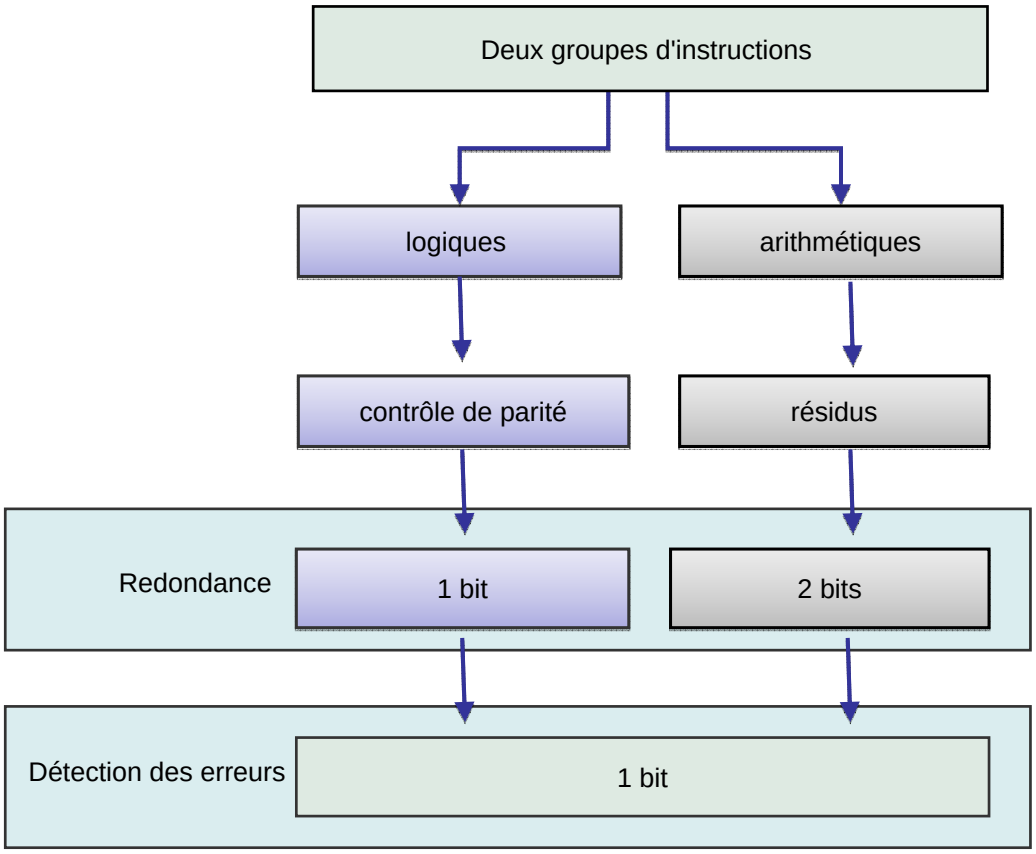


FIGURE 4.12 – Protections distinctes des instructions logiques et arithmétiques

où PA_X and PA_Y représentent les restes de X et Y, respectivement, et $Rx_m = x_n/C$ et ainsi de suite.

La sortie de l'ALU est représentée comme suit :

$S = X \bigoplus Y$ où $\bigoplus$ = `ADD`, `SUB` ou `MUL`

$PA'_S = S \mod C$

PA_S est le symbole de contrôle du reste qui est donnée par :

$PA_S = (PA_X \bigoplus PA_Y) \mod C$

Le signal d'erreur est généré par le comparateur, ce qui est donné par la fonction suivante :

Signal d'erreur $= 1$ si $PA_S \neq PA'_S$

Signal d'erreur $\neq 1$ si $PA_S = PA'_S$

PL_S, représente la parité logique qui sera générée localement pour l'instruction suivante.

Par exemple, si $X = 10$, $Y = 11$ et $C = 3$

Résidu de X (PA_X) : $10 \mod 3 = 1$

Résidu de Y (PA_Y) : $11 \mod 3 = 2$

Premier calcul concurrent : $S = X + Y = 10 + 11 = 21$

Résidu du premier calcul : $PA'_S = 21 \mod 3 = 0$

Addition de PA_X et PA_Y : $1 + 2 = 3$

Résidu : $PA_S = 3 \mod 3 = 0$

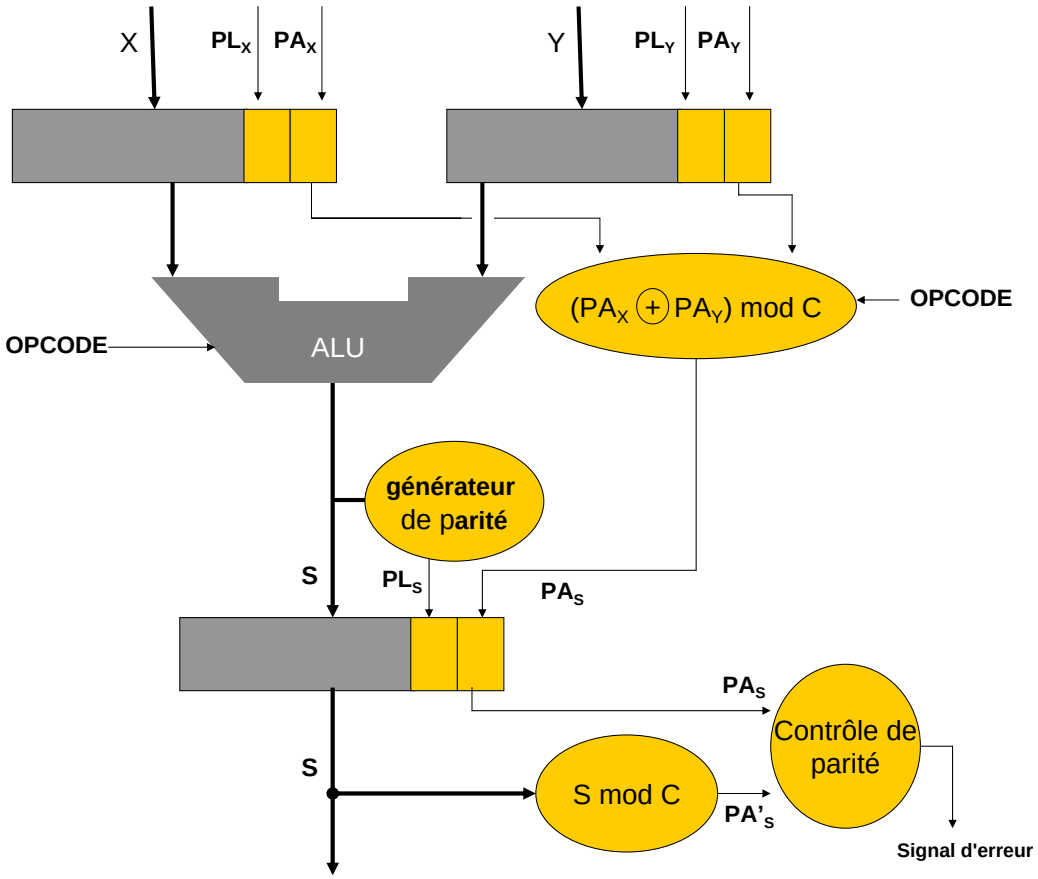


FIGURE 4.13 – Technique de contrôle du reste pour la détection d'erreurs dans les instructions arithmétiques

Ainsi, les résidus PA_S et PA'_S sont égaux, il n'y a donc pas d'erreur.

Détection d'erreurs dans les instructions logiques

La détection des erreurs dans les instructions logiques est basée sur le calcul d'un bit de parité à partir des symboles d'information X et Y. Le calcul de la parité est simple. Le bit de parité est calculé par l'opération XOR entre les bits d'information. Parmi les deux variantes du bit de parité – bit de parité paire ou bit de parité impaire – utiliser une parité paire signifie que le bit de parité est mis à 1 si le nombre de uns dans un ensemble donné de bits (non compris le bit de parité) est impair, ce qui rend l'ensemble des bits (y compris le bit de parité), pair. En comparant les bits de parité en entrée et en sortie, le signal de régénération/reconfiguration est positionné au niveau haut ou bas.

Ceci peut être représenté par la simple équation logique suivante :

$PL_X = x_{15}$ XOR x_{14} XOR x_{13} XOR ... x_0

$PL_Y = y_{15}$ XOR y_{14} XOR y_{13} XOR ... y_0

De même, PL_X et PL_Y représentent la parité de X et Y, respectivement.

$S = (X \odot Y)$ où $\odot$ = AND ou OR

où $X = (x_{15}\ x_{14}\ x_{13}\ ...\ x_0)$ et $Y = (y_{15}\ y_{14}\ y_{13}\ ...\ y_0)$

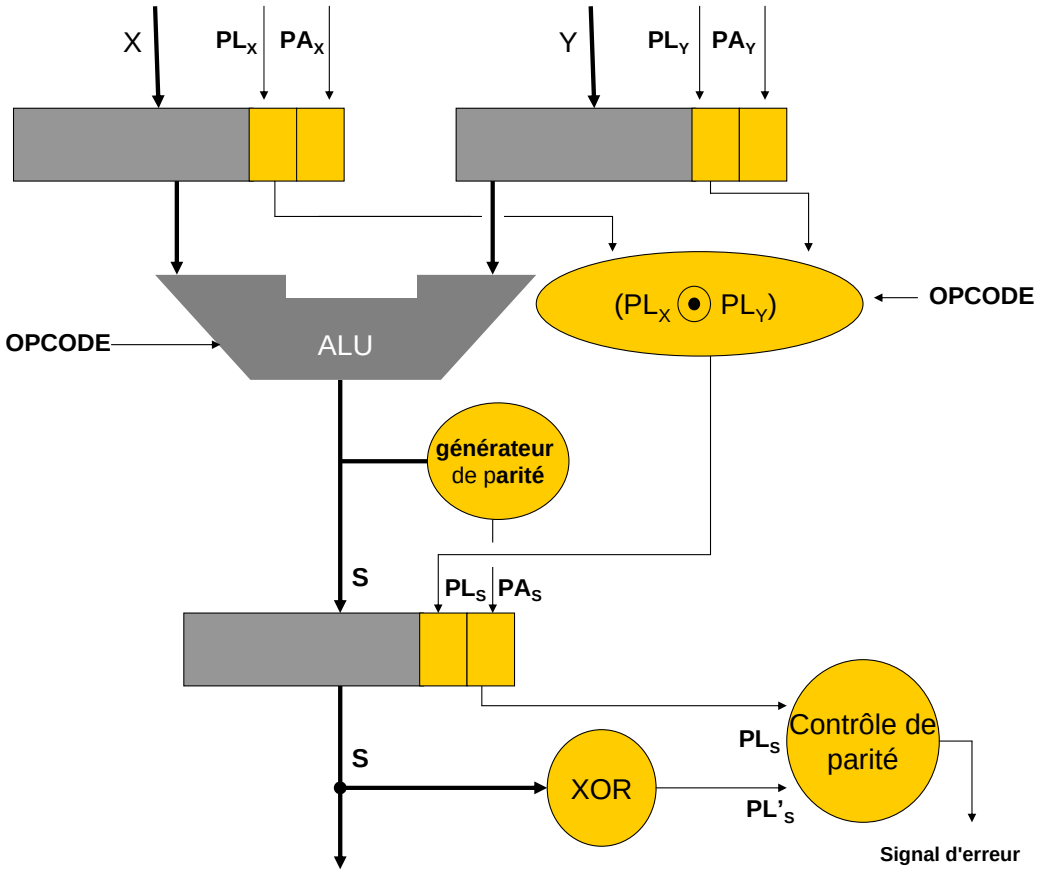


FIGURE 4.14 – Technique de contrôle de parité pour la détection des erreurs dans les instructions logiques

$PL_S = PL_X \odot PL_Y$

$PL'_S = S$ XOR

Le signal d'erreur est généré par le comparateur, et donné par la fonction suivante :

Signal d'erreur $= 1$ si $PL_S \neq PL'_S$

Signal d'erreur $\neq 1$ si $PA_S = PA'_S$

De même, PA_S sera généré localement pour l'instruction suivante (si nécessaire). C'est un système synchrone et l'erreur sera détectée au cycle d'horloge suivant. Dans l'ALU, la latence d'erreur est de 1 cycle d'horloge.

4.5.2 Détection d'erreurs dans les registres

Pour la détection d'erreurs dans les registres, nous nous appuyons également sur les codes de parité. Les registres contrôlent de manière concurrente les erreurs en comparant les bits de parité générés et ceux déjà existants. En cas d'inégalité, un signal d'erreur est positionné (comme le montre la figure 4.15). Par ailleurs, un contrôle de parité simple peut uniquement détecter les erreurs bit uniques ou les erreurs dont la multiplicité est impaire.

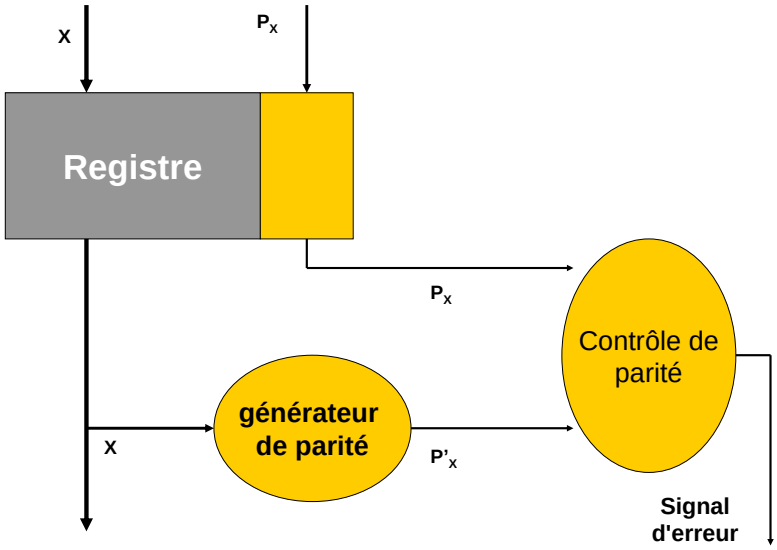


FIGURE 4.15 – Technique de contrôle de parité pour la détection des erreurs dans les registres

4.5.3 Processeur autocontrôlé

Avec les protections présentées dans les sections 4.5.1 et 4.5.2, le processeur tolérant aux fautes intègre des mécanismes d'autocontrôle permettant de détecter les SBU. La couverture d'erreur peut être améliorée par des EDC supplémentaires, ce qui augmente cependant la complexité du circuit.

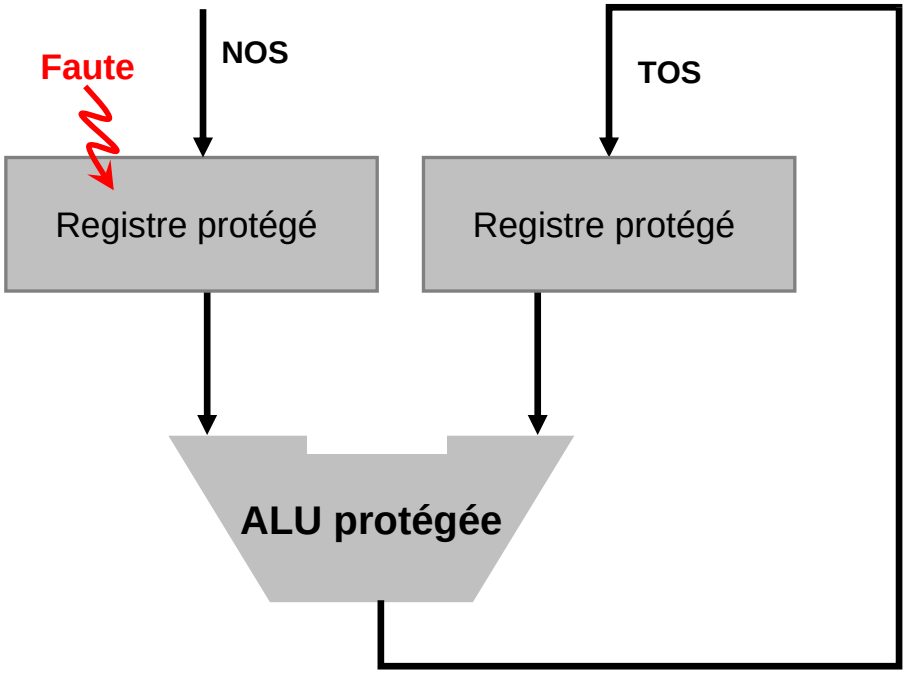


FIGURE 4.16 – Erreur se produisant dans une ALU protégée

4.5.4 Sauvegarde des éléments sensibles (SE)

Les six états internes TOS, NOS, TORS, DSP, RSP, et IP doivent être sauvegardés à la fin d'une séquence valide pour permettre d'éventuels *rollbacks*. Nous avons décidé de les stocker dans la mémoire sûre. La procédure suit six instructions consécutives où le contenu de TORS est stocké dans la pile de retour, RS et les autres dans la pile de donnée, DS. L'aspect positif de cette approche est qu'elle n'implique aucune surcharge matérielle supplémentaire à l'intérieur du processeur, l'inconvénient étant la perte de performances due au stockage des SE. Une combinaison possible des instructions peut être :

```
CALL a
CPR2D
PUSH_RSP
PUSH_DSP
DUP
DUP
```

Une solution alternative pour protéger les états internes est l'utilisation de registres copies internes conservant les valeurs présentes en fin de la précédente séquence valide. En cas de *rollback*, les contenus des registres copies sont chargés dans les registres correspondants. L'avantage de ce schéma est qu'un seul cycle d'horloge est nécessaire pour sauver ou restaurer les registres. Toutefois, cela doublerait le nombre de registres des SE, et ces registres copies doivent également être protégés, ce qui entraîne un surcoût matériel supplémentaire. Par ailleurs, ce n'est pas une solution favorable pour le changement de contexte.

4.5.5 Protéger les opcodes

La mémoire programme est incluse dans la mémoire principale sûre, par conséquent il n'y a aucun risque de fautes, mais elles peuvent survenir à l'intérieur de l'opcode lors de l'exécution. Heureusement, une protection de l'opcode sans pénalité matérielle est possible, parce que de opcode de 6 bits permettent de définir 64 instructions différentes. Or, nous avons choisi des opcodes de 8 bits pour seulement 37 instructions, ce qui laisse 2 bits disponibles pour mettre en œuvre des EDC à faible surcoût.

4.6 Solution n°II : Aspects performance du cœur du processeur

Grâce à l'architecture à pile choisie, les données sont implicitement disponibles sur les deux sommets de la pile, ce qui réduit la longueur du chemin critique. Mais pour atteindre des performances (temporelles) élevées, (i) le taux moyen d'exécution des instructions (nombre de cycles d'horloge par instruction) doit être proche de l'unité, et (ii) pour les instructions multi-octets, l'instruction suivante doit parvenir à l'unité de contrôle au cycle d'horloge $(t + 1)$. En d'autres termes, il doit y avoir un flot continu d'instructions dans le buffer d'instructions IB. L'unité de gestion du buffer d'instructions (IBMU) est dédiée à cette tâche.

L'IBMU génère six signaux de commande différents, parmi lesquels cinq sont dédiés au contrôle du flot de données dans l'IB, à savoir SM1, SM2, SM3, SM4 et SM5, comme indiqué dans la figure 4.17, tandis que le sixième signal (SM6) est réservée au pointeur d'instruction IP. La section suivante traite des solutions proposées pour chacun d'entre d'eux.

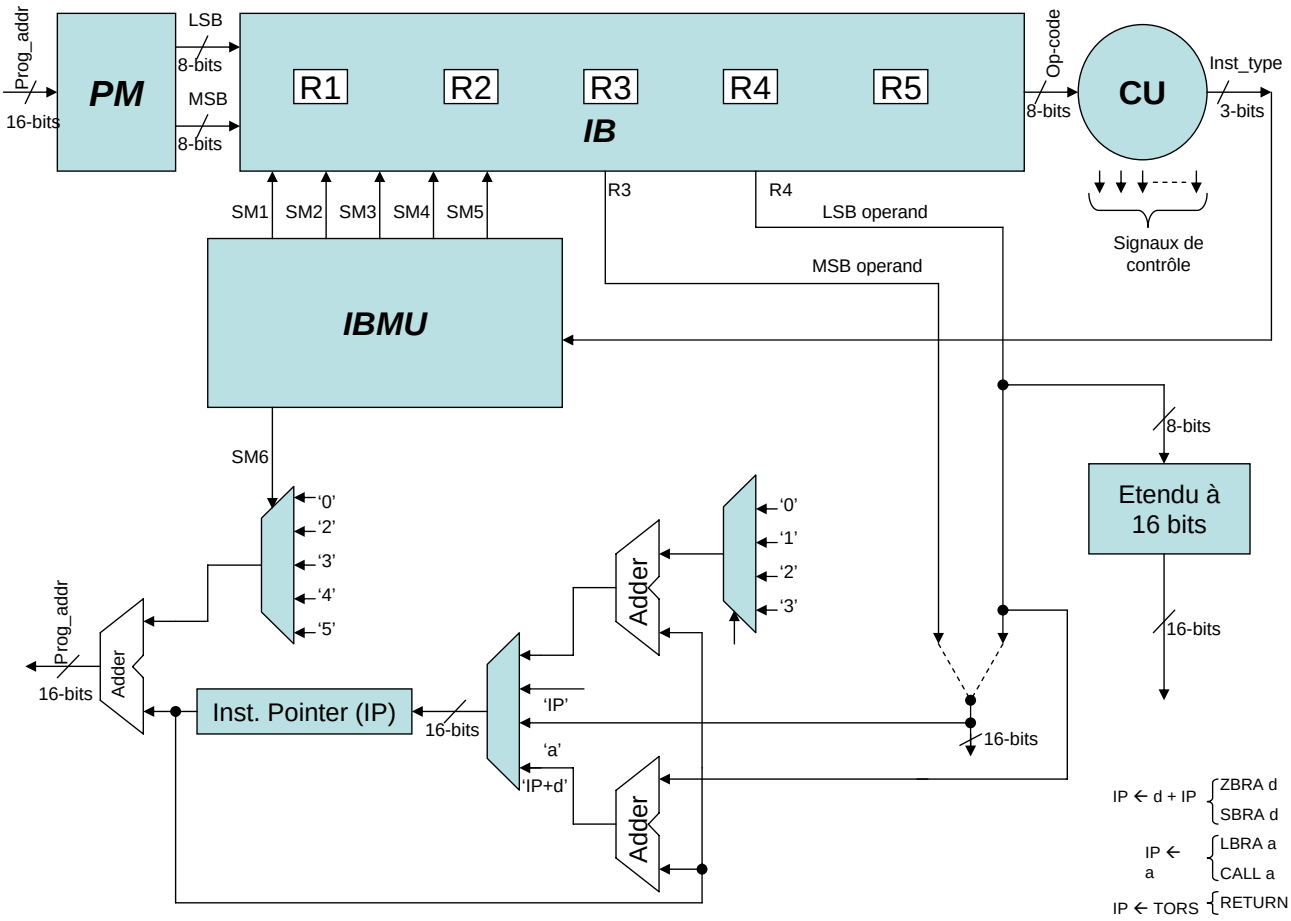


FIGURE 4.17 – Unité de gestion du buffer d'instructions (IBMU)

4.6.1 Solution n°II.a : instructions multi-octets

Il y a sept instructions de plusieurs octets, constituées de blocs de 2 ou 3 octets. L'IBMU contrôle le flot d'instructions dans le buffer en pré-chargeant la prochaine instruction à exécuter et en la rendant disponible dans l'unité de contrôle au cours du cycle d'horloge suivant $(t + 1)$. L'IBMU contrôle la série de buffers en cascade ayant des interconnexions multiples pour faire face à des conditions complexes, comme indiqué dans la figure 4.18. Les décisions à l'intérieur de l'IBMU sont prises en fonction des états prédéfinis de la machine d'états (FSM, *Finite State Machine*). Les transitions entre ces états dépendent de l'état actuel du buffer d'instructions.

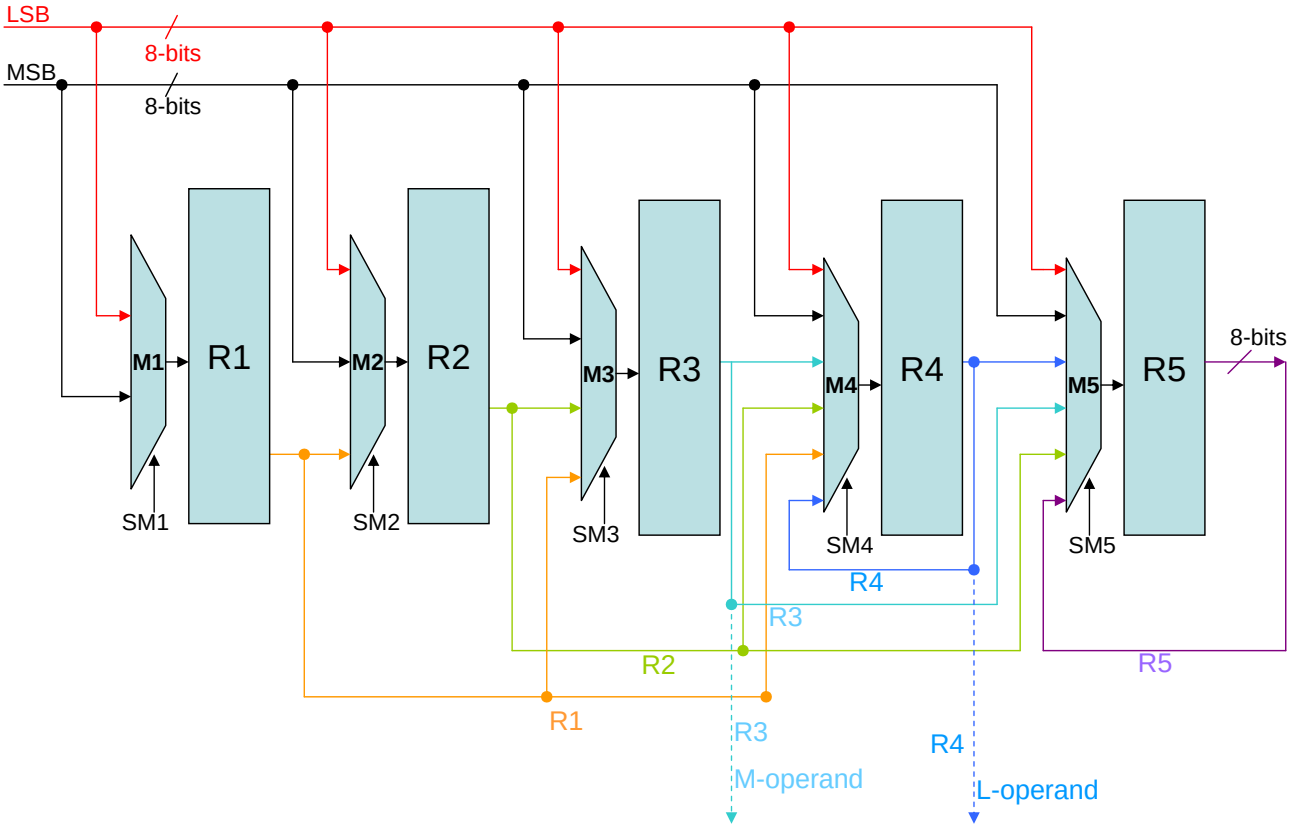


FIGURE 4.18 – Buffer d'instructions (IB)

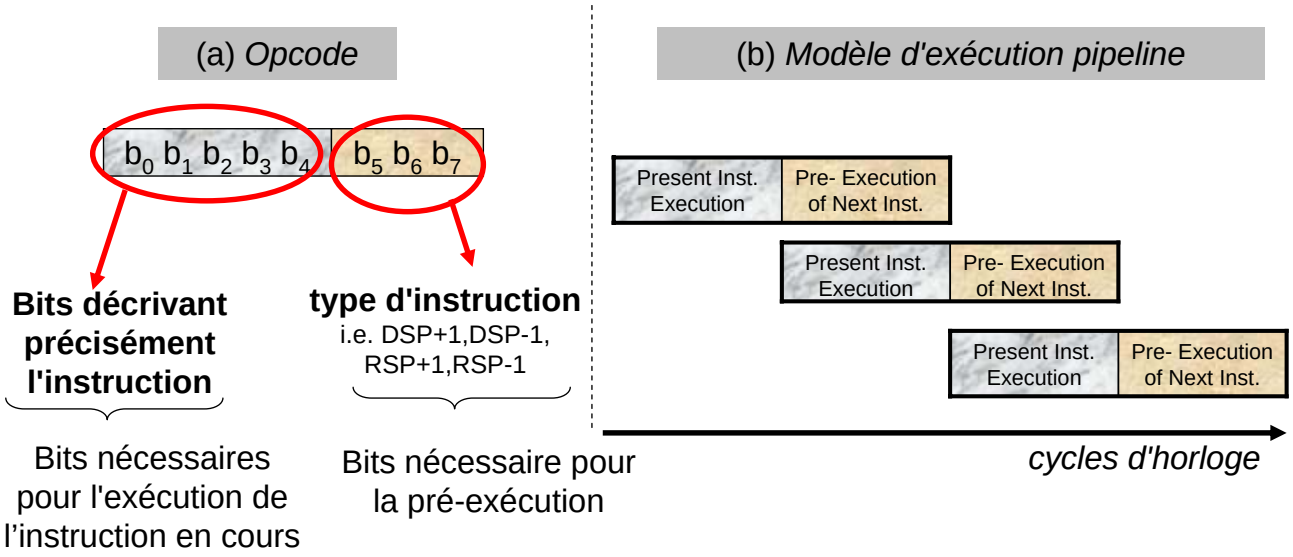


FIGURE 4.19 – (a) description des opcodes ; (b) modèle d'exécution en pipeline

4.6.2 Solution n°II.b : pipeline à 2 étages pour résoudre l'exécution des instructions multi-cycles

La majorité des instructions nécessitent un seul cycle d'horloge pour être exécutées, alors que d'autres nécessitent un nombre de cycles d'horloge multiple. Pour exploiter correctement le pipeline,

TABLE 4.1 – Types d'instructions

b7 b6 b5	Détails
1 0 0	1 octet
1 1 1	1 octet (multi-cycles)
1 0 1	2 octets
1 1 0	3 octets
0 1 1	1 octet + modif. IP
0 0 1	2 octets + modif. IP
0 1 0	3 octets + modif. IP

nous devons différencier les instructions à cycle unique de celles à cycles multiples. Les trois bits les plus significatifs (b_7, b_6, et b_5) de l'opcode sont réservés pour déterminer le type de l'instruction comme indiqué sur la figure 4.19 (a). En effet, les instructions qui nécessitent plusieurs cycles d'horloge pour être exécutées se sont vu attribuer le code « 111 ». Nous pouvons distinguer les différentes instructions sur la base de la longueur de l'instruction et de la modification du contenu du registre IP, comme indiqué dans le tableau 4.1. Le changement du pointeur IP intervient dans les instructions de saut.

Les instructions à cycles d'horloge multiples ont été analysées – avec des combinaisons différentes d'instructions – afin de mettre en évidence les éventuels conflits entre elles. Nous en avons conclu que si elles sont exécutées dans un pipeline à deux étages, alors tous les conflits d'adressage de la mémoire peuvent être évités. Dans la première phase, les pointeurs de piles sont incrémentés ($DSP+1/RSP+1$) en fonction du type d'instruction, alors que dans la deuxième phase le reste de l'instruction est exécuté ; il n'y aura alors pas de conflit lors de l'accès à la mémoire. Les registres DSP (*Data Stack Pointer*) et RSP (*Return Stack Pointer*) pointent en mémoire à DS et RS respectivement. De cette analyse résulte l'exécution pipelinée à deux étages, comme indiqué dans la figure 4.19 (b).

Lors du pipelining, une partie de l'instruction suivante est pré-exécutée ($DSP + 1/RSP + 1$), en même temps que l'instruction active, en un cycle d'horloge. Ainsi, la partie restante de l'instruction peut être exécutée en un seul cycle également, au cycle d'horloge suivant. L'unité de contrôle utilise les 8 bits de l'opcode de l'instruction en cours pour générer les signaux de commande pour tous les multiplexeurs associés. Simultanément, les trois bits de poids fort de l'opcode de la prochaine instruction sont également extraits, parce que ces MSB identifient le type de l'instruction.

Pour évaluer l'efficacité du pipeline, nous avons réalisé un benchmark composé de cinq instructions (indiquées dans 4.20). Sans pipeline, ce programme nécessite 9 cycles d'horloge, alors que 5 seulement sont nécessaires avec le pipeline, ce qui conduit à une amélioration de 45 %.

Ainsi, grâce au pipeline, toutes les instructions peuvent être exécutées en un seul cycle de contrôle, à l'exception de l'instruction `STORE` qui en nécessite deux. De fait, l'instruction `STORE` nécessite d'incrémenter deux fois le pointeur de pile ($DSP + 1$), ce qui ne peut être fait en un seul cycle d'horloge. La liste complète des instructions est donnée dans les tableaux en annexe C.

• Non-Pipeliné

Cycle	Operation	Instruction exécutée
1	TOS ← TOS + NOS NOS ← DS [DSP]	ADD
2	DSP ← DSP +1	DUP
3	DS [DSP] ← NOS NOS ← TOS	
4	DSP ← DSP +1	R2D
5	TOS ← TORS NOS ← TOS DS [DSP] ← NOS TORS ← RS [RSP] RSP ← RSP -1	
6	DSP ← DSP +1	PUSH_DSP
7	DS [DSP] ← NOS NOS ← TOS TOS ← DSP	
8	DSP ← DSP +1	LITa
9	DS[DSP] ← NOS NOS ← TOS TOS ← data (byte)	

• Pipeliné

Cycle	1er étage		2eme étage		Instruction exécutée
	Instruction	Operation	Instruction	Operation	
1	DUP	DSP ← DSP +1	ADD	TOS ← TOS + NOS NOS ← DS [DSP]	ADD
2	R2D	DSP ← DSP +1	DUP	DS [DSP] ← NOS NOS ← TOS	DUP
3	PUSH_DSP	DSP ← DSP +1	R2D	TOS ← TORS NOS ← TOS DS [DSP] ← NOS TORS ← RS [RSP] RSP ← RSP -1	R2D
4	LIT a	DSP ← DSP +1	PUSH_DSP	DS [DSP] ← NOS NOS ← TOS TOS ← DSP	PUSH_DSP
5		NOP	LIT a	DS[DSP] ← NOS NOS ← TOS TOS ← data (byte)	LIT a

FIGURE 4.20 – Exemple de programme exécuté par le processeur à pile sans pipeline et avec pipeline

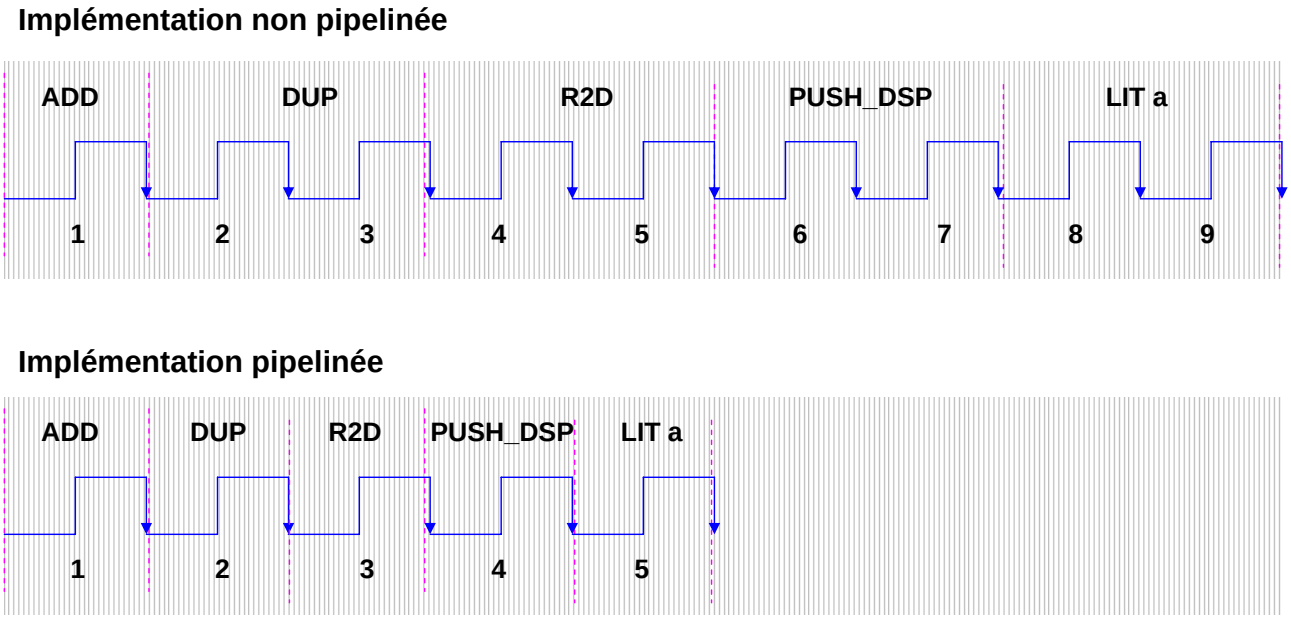


FIGURE 4.21 – Chronogrammes d'un même programme exécuté deux fois : d'abord sans pipeline, ensuite avec pipeline

4.6.3 Réduction de la pénalité due aux branchements conditionnels

Il a précédemment été indiqué que le chargement des instructions dans le buffer IB est presque deux fois plus rapide que leur exécution, les opcodes de 8 bits étant exécutés en un cycle d'horloge. Par conséquent, l'instruction suivante est déjà chargée dans le buffer. Toutefois, en cas d'instruction

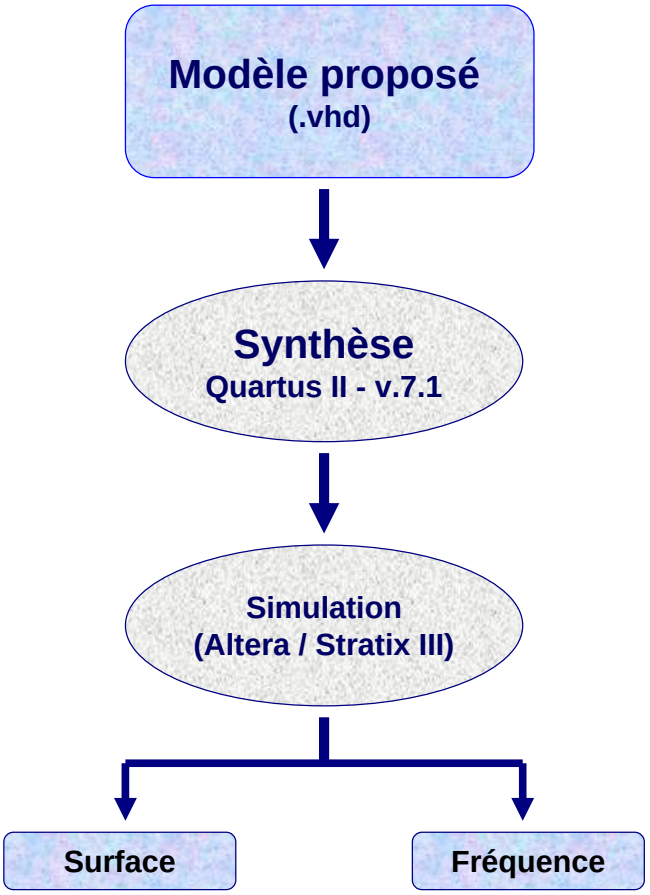


FIGURE 4.22 – Flot de conception

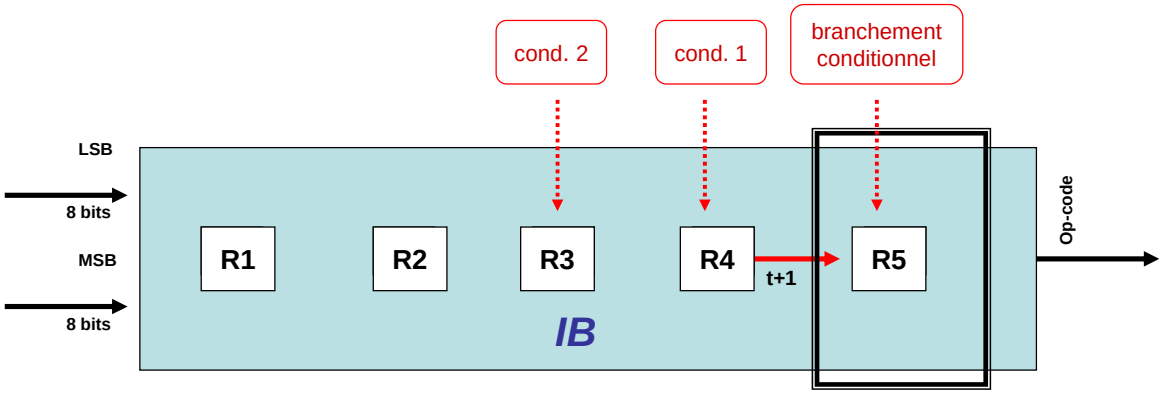


FIGURE 4.23 – Stratégie pour surmonter la pénalité de performance due aux branchements conditionnels

de saut, l'IB doit être vidé et de nouvelles instructions doivent être chargées. Il peut en résulter des pénalités de performance. Ceci peut être surmonté si nous profitons du fait que le chargement des instructions est plus rapide que leur consommation par le cœur du processeur à pile.

L'approche sera basée sur le chargement des deux conditions de saut à l'intérieur de l'IB (comme le montre la figure 4.23. Par conséquent, il n'y aura pas d'instruction `NOP` insérée après une instruction

de saut. Cependant, cela peut augmenter la complexité de l'unité de gestion du buffer d'instructions, et peut nécessiter un buffer d'instructions plus large. L'implantation VHDL au niveau RTL d'une telle solution n'est pas considérée dans ce travail.

4.7 Résultats d'implantation

Le processeur autocontrôlé a été synthétisé avec Quartus II d'Altera. La figure 4.22 montre le flot de conception du SCPC modélisé en VHDL au niveau RTL et implanté sur un Altera Stratix III EP3SE50F484C2 à l'aide de Quartus II. À partir des résultats, les observations suivantes peuvent être faites :

- *Analyse de la surface :* les résultats obtenus en termes de surface sont rapportés dans le tableau 4.2. La surface nécessaire pour le SCPC est minimale, ce qui en fait un processeur adapté au développement d'un futur MPSoC.

TABLE 4.2 – Surface occupée par le processeur

	Comb. ALUT	Logique dédiée
SCPC	861	278

- *Analyse de la performance :* bien que dans ce chapitre nous avons uniquement modélisé un cœur de processeur (SCPC) et que le modèle doive être complété par un l'implantation du journal matériel autocontrôlé (SCHJ) – ce que nous étudierons dans les prochains chapitres, les aspects de performance du processeur peuvent être analysés pour connaître l'efficacité de l'approche à pile. Dans un processeur à pile, nous avons un cycle d'horloge court car les opérandes sont implicitement disponibles sur les deux sommets de la pile. Il est intéressant de noter que le processeur à pile choisi nécessite deux étages de pipeline pour obtenir de bonnes performances. Toutes les instructions (à l'exception de `STORE`) peuvent être exécutées en seul cycle d'horloge. La performance de l'architecture a été validée et les résultats sont présentés dans la figure 4.24. Les résultats montrent l'exécution des instructions en un seul cycle d'horloge.

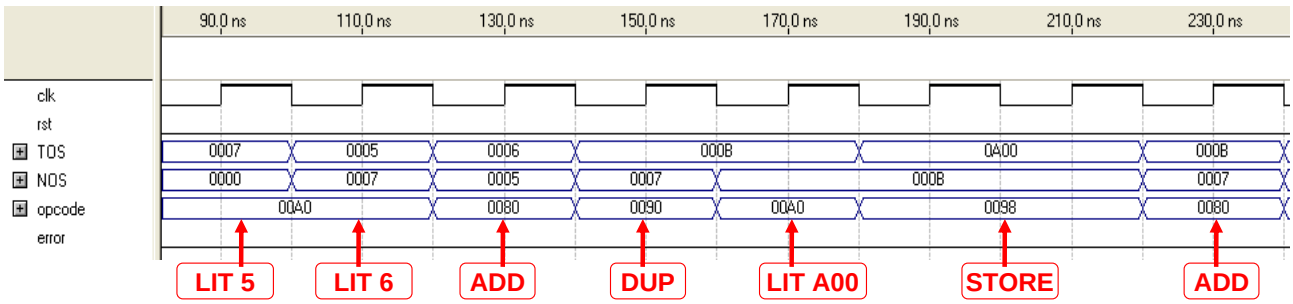


FIGURE 4.24 – Implantation du processeur autocontrôlé

- *Analyse de l'autocontrôle :* nous allons valider la capacité de détection d'erreur en injectant des erreurs simples (SBU). Toutefois, la validation complète du modèle global sera présentée dans le chapitre 6, où différents scénarios d'erreur seront mis en œuvre pour l'injection artificielle d'erreurs, afin de vérifier l'efficacité de l'approche globale. Ici, les résultats d'implantation de la figure 4.25 montrent le processeur en mode lecture/écriture (mode 01). Les différents modes de travail du processeur seront discutés dans le chapitre 5. À un instant déterminé par le scénario, une erreur est artificiellement injectée dans le processeur autocontrôlé. Lors de la détection d'une erreur, l'exécution des instructions est stoppée et le processeur revient en arrière (*rollback*).

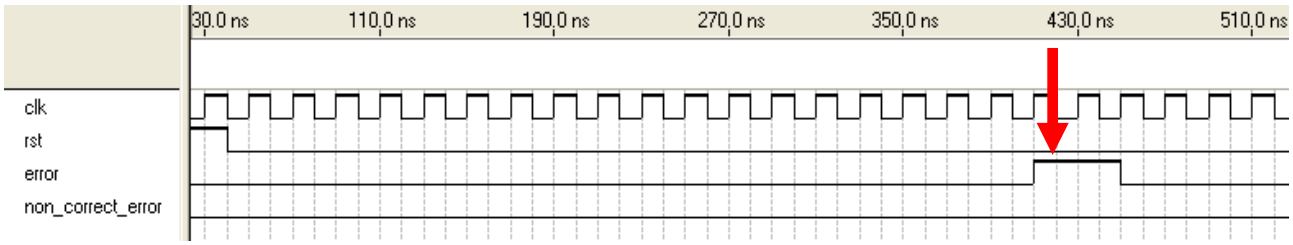


FIGURE 4.25 – Erreur détectée dans le SCPC

4.8 Conclusion

Dans ce chapitre, nous avons conçu un cœur de processeur autocontrôlé (SCPC) présentant une tolérance contre les SBU, et défini les mesures à prendre pour en améliorer les performances. Les choix de conception ont été faits en vue d'assurer une détection d'erreur rapide avec un surcoût matériel minimal. La détection d'erreur est basée sur les codes combinatoires (résidu et parité), alors que le recouvrement d'erreur est basé sur un mécanisme de rétroaction (*rollback*).

Le point intéressant est le choix d'un processeur à pile de classe MISC. Il s'agit d'un processeur simple ayant un nombre réduit d'états internes, ce qui est favorable tant pour la protection par CED que pour le *rollback*. Il occupe une faible surface sur la puce, ce qui constitue un avantage certain du point de vue de la fiabilité et de l'économie de matériel.

Pour améliorer la vitesse d'exécution des instructions, le processeur intègre un pipeline d'exécution à deux étages. L'unité de gestion du buffer d'instructions contrôle le flot d'instructions multi-octets dans le buffer. Par conséquent, nous profitons de la densité de code élevée des instructions à longueur variable tout en permettant l'exécution en deux phases dans le pipeline, dans lequel une part de l'instruction suivante est pré-exécutée en même temps que l'instruction en cours.

Dans le prochain chapitre, nous allons discuter de la conception et de l'implantation du journal matériel autocontrôlé qui nous permettra de masquer les erreurs et d'empêcher leur propagation vers la DM.

Chapitre 5

Conception du journal matériel autocontrôlé

Ce chapitre est consacré à la conception du journal matériel autocontrôlé (*Self Checking Hardware Journal*, SCHJ) qui est utilisé comme élément central de notre stratégie de conception d'un processeur tolérant aux fautes contre les fautes transitoires (comme le montre la figure 5.1).

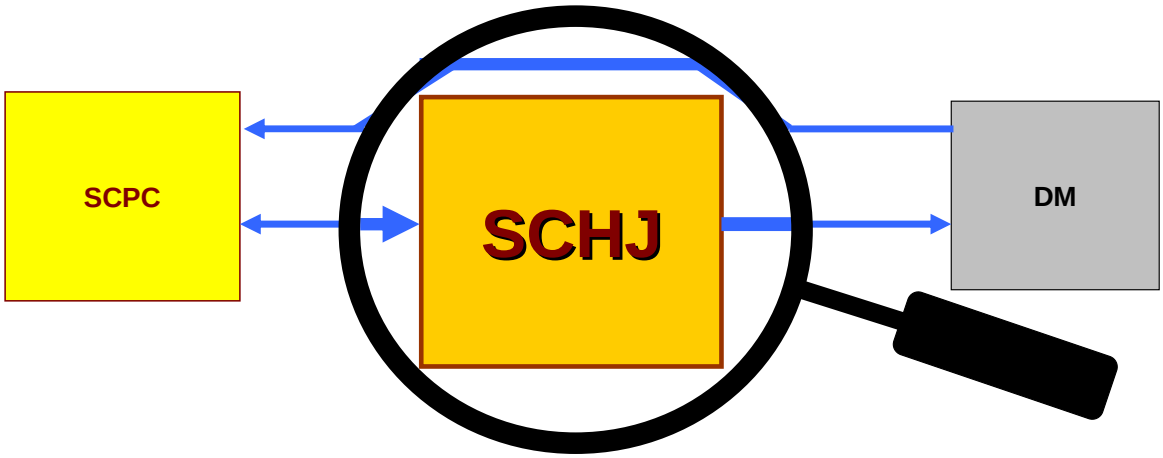


FIGURE 5.1 – Conception du SCHJ

Le rôle fondamental de ce SCHJ est de contenir les nouvelles données générées pendant la séquence en cours d'exécution jusqu'à ce qu'elle soit validée (voir figure 5.2). Si la séquence est validée, ces données peuvent être transférées à la mémoire sûre. Sinon, en cas de détection d'erreur au cours de la séquence actuelle, ces données sont tout simplement ignorées et la séquence en cours est relancée depuis le début en rechargeant les données fiables conservées en mémoire et correspondant à l'état prévalant à la fin de la séquence précédente. Cependant, nous avons besoin d'un mécanisme de détection et de correction d'erreur intégré au journal afin de détecter les erreurs potentielles sur les données lors de leur séjour temporaire dans celui-ci.

Ce chapitre étudie la construction et le fonctionnement du SCHJ et le travail est réparti comme suit. La première section présente la méthodologie d'autocontrôle. Dans la section suivante, l'architecture et le fonctionnement du journal sont décrits. Enfin, pour évaluer le fonctionnement du journal matériel autocontrôlé, un modèle générique est décrit en VHDL au niveau RTL puis synthétisé à l'aide

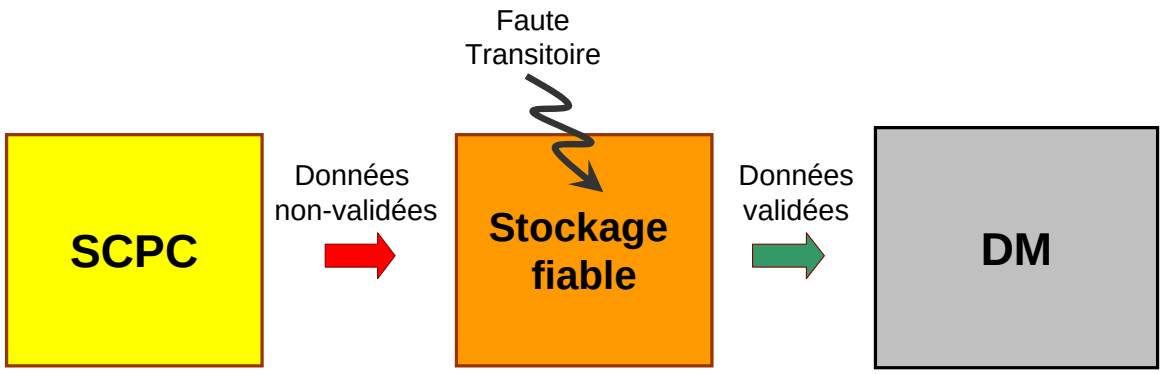


FIGURE 5.2 – Protection de la mémoire contre la contamination.

de Quartus II d'Altera.

5.1 Détection et correction des erreurs dans le journal

Il a été montré dans la section 3.4.2 que le journal devait intégrer un mécanisme d'autocontrôle, car les données qui y sont temporairement stockées peuvent également être corrompues en conséquence à des fautes transitoires affectant celui-ci (voir les figures 5.2 et 5.3).

Le journal contiendra d'une part des données appartenant à la séquence actuelle dans sa partie non validée (partie supérieure de la figure 5.3 (a)), et d'autre part – dans sa partie validée – des données appartenant à la précédente séquence (partie inférieure de la figure 5.3 (b)). Si une erreur est détectée dans les données appartenant à la séquence en cours, alors nous pouvons procéder à la restauration des précédents états validés. Toutefois, si une erreur survient dans les données qui ne correspondent plus à l'état actuel du processeur, nous ne pouvons pas mettre en œuvre le *rollback* parce que les états des éléments sensibles du processeur ne sont plus présents en mémoire comme montré dans la figure 5.3 (b). Cela signifie que seule une détection d'erreur est nécessaire dans la partie non validée du journal (UVJ), mais la correction d'erreur est indispensable, en plus de la détection, dans sa partie validée (VJ).

Un ECC sera employé pour la détection et la correction des erreurs dans le SCHJ. Les codes de Hamming et de Hsiao sont les plus couramment employés [Sta06] dans les mémoires. Le code de Hsiao est plus efficace et implique un surcoût matériel inférieur aux codes de Hamming [GBT05]. Il a été largement utilisé dans la conception de mémoires fiables [Che08]. Bien qu'employé depuis trois décennies, il reste le code le plus efficace utilisé dans l'industrie [GBT05, Che08]. La protection du journal sera donc confiée au code de Hsiao.

5.2 Principe de la technique

Le code de Hsiao [Hsi10] permet un encodage rapide, ainsi qu'une détection d'erreur rapide lors du processus de décodage. Il est obtenu à partir d'un raccourcissement des codes de Hamming. La

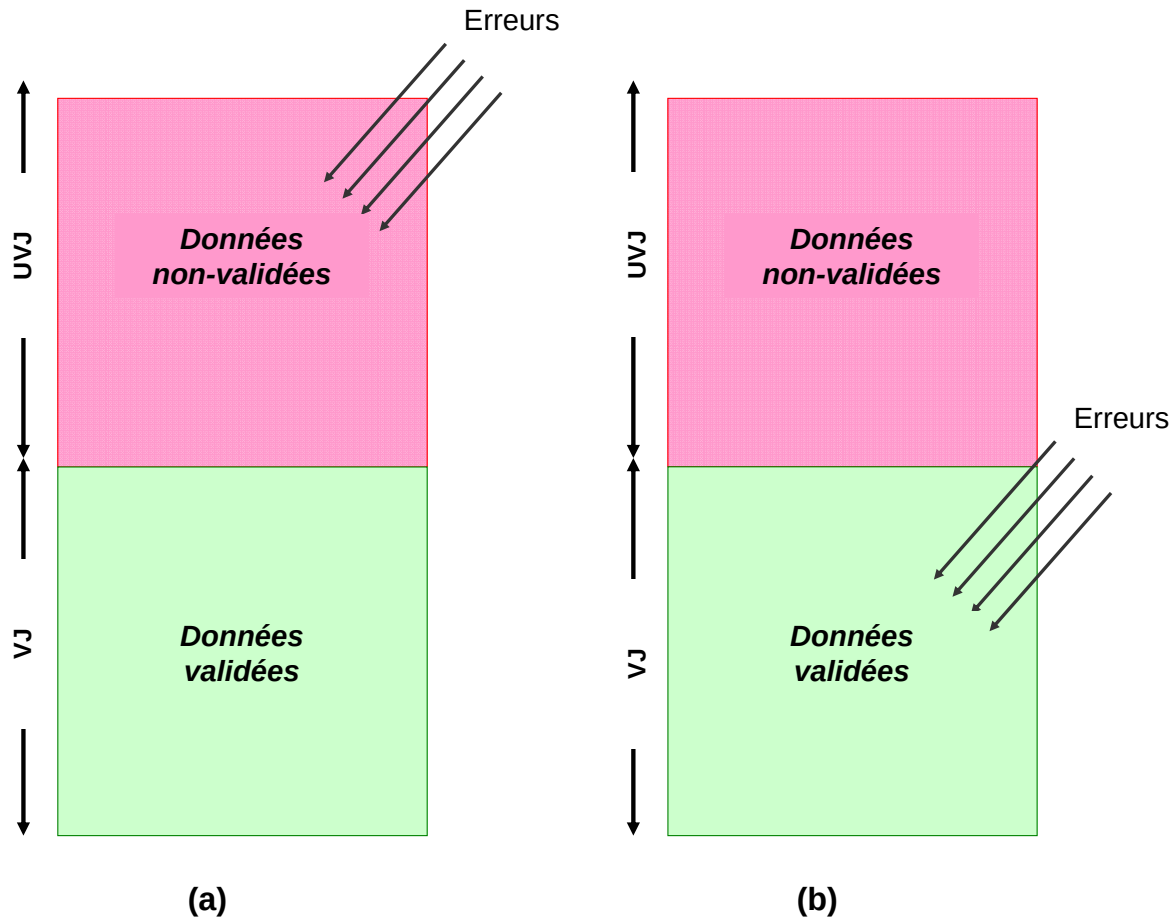


FIGURE 5.3 – (a) Erreur(s) dans l'UVJ (b) erreur(s) dans le VJ

construction du code est mieux décrite en termes de matrice de parité (H_0). La sélection des colonnes de la matrice H_0 pour un code donné (n, k) est basée sur trois conditions :

- chaque colonne doit avoir un nombre impair de 1 ;
- le nombre total de 1 dans la matrice H_0 doit être minimal ;
- Le nombre de 1 dans chaque ligne de H_0 doit être égal ou le plus proche possible du nombre moyen, c'est-à-dire du nombre total de 1 dans H_0 divisé par le nombre de lignes.

La première condition garantit que le code généré par H_0 a une distance minimale d'au moins 4. Donc, il peut être utilisé pour une correction d'erreur simple et la détection d'erreurs doubles. Les deuxièmes et troisièmes conditions fixent les niveaux logiques minimaux pour les bits de parité (syndrome), et permettent de minimiser le matériel nécessaire à la mise en œuvre du code.

Par exemple, si r bits de contrôle de parité sont utilisés pour correspondre aux k bits de données, alors l'équation suivante doit être vraie pour les codes de Hsiao :

$$\sum_{i=1, i=odd}^{\leq r} \begin{pmatrix} r \\ i \end{pmatrix} \geq r + k \tag{5.1}$$

Plus précisément, la matrice H_0 est construite comme suit :

(a) toutes les colonnes de poids 1 $\binom{r}{1}$ sont utilisées pour les positions des r bits de contrôle de parité ;

(b) ensuite, si $\binom{r}{3} \geq k$, on sélectionne k colonnes de poids 3 parmi toutes les $\binom{r}{3}$ combinaisons possibles ; si $\binom{r}{3} < k$, toute colonne $\binom{r}{3}$ de poids 3 doit être sélectionnée. Le reste des colonnes sont tout d'abord choisies parmi toutes celles de poids 5 $\binom{r}{5}$, puis parmi celle de poids 7 $\binom{r}{7}$, et ainsi de suite jusqu'à ce que toutes les k colonnes aient des combinaisons uniques.

Si la longueur du mot code $n = k + r$ est exactement égale à

$$\sum_{i=1,i=odd}^{\leq r} \begin{pmatrix} r \\ i \end{pmatrix} \tag{5.2}$$

pour certains $j \leq r$ impairs, chaque ligne de la matrice H_0 aura le nombre de 1 suivant :

$$\begin{aligned} \frac{1}{r} \sum_{\substack{i=1 \\ i=odd}}^{i \leq r} i \begin{pmatrix} r \\ i \end{pmatrix} &= \frac{1}{r} \left[r + 3\frac{r(r-1)(r-2)}{3!} + \cdots + j\frac{r(r-1)\cdots(r-j+1)}{j!} \right] \\ &= \left[1 + \begin{pmatrix} r-1 \\ 2 \end{pmatrix} + \cdots + \begin{pmatrix} r-1 \\ j-1 \end{pmatrix} \right] \end{aligned} \tag{5.3}$$

Si n n'est pas exactement égal à $\sum_{i=1,i=odd}^{\leq r} \begin{pmatrix} r \\ i \end{pmatrix}$ pour certains j, alors la sélection arbitraire des cas $\binom{r}{i}$ doit permettre d'obtenir un nombre de 1 dans chaque ligne proche de la moyenne.

La correction d'erreur unique et la détection d'erreurs doubles sont réalisées de la façon suivante. Une erreur unique résulte en un motif de syndrome qui doit correspondre à une colonne de la matrice de contrôle de parité H_0. Ainsi, en faisant correspondre un motif de syndrome à une colonne dans H_0 on peut identifier un bit erroné. Si la colonne correspond à un bit de contrôle, alors aucune correction n'est nécessaire [Lal05], sinon une inversion du bit permet de corriger l'erreur. La détection des erreurs doubles est accomplie par l'examen de la parité d'ensemble de tous les bits du syndrome. Comme le code Hsiao utilise un nombre impair de 1 dans les colonnes de sa matrice H_0, un motif de syndrome correspondant à une erreur bit unique possède une parité impaire. En revanche, si elle possède un nombre pair de bits, cela indique alors la présence d'une erreur double dans un mot.

Hsiao a montré qu'en utilisant des colonnes de poids impaire minimal, le nombre de 1 dans la matrice H_0 peut être minimisé, et être inférieur à celui d'un code de Hamming. Cela se traduit par moins de ressources matérielles nécessaires dans les circuits ECC correspondants. Par ailleurs, en sélectionnant les colonnes de poids impair d'une manière qui équilibre le nombre de 1 à chaque ligne de la matrice H_0, le retard dû au contrôleur de parité peut être minimisé, puisque ce retard est contraint par la ligne de poids maximal.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	C1	C2	C3	C4	C5	C6	C7
S1	1	1	1	1	1	1	1	1							1					1			1		1						1	1			1						
S2					1			1	1	1	1	1	1	1	1	1			1			1			1					1			1			1					
S3				1								1					1	1	1	1	1	1	1	1			1	1		1	1		1				1				
S4			1				1				1			1		1	1									1	1	1	1	1	1	1		1				1			
S5		1	1			1		1		1			1			1					1	1	1	1		1	1		1				1						1		
S6	1					1	1		1				1	1	1		1	1	1	1	1								1					1						1	
S7	1	1		1	1				1	1	1	1						1						1		1		1				1		1						1

FIGURE 5.4 – Matrice de contrôle de parité matrice (41,34) de Hsiao

En pratique, les données résidant dans le SCHJ sont codées avec des bits de parité générés selon le code Hsiao [Hsi10]. Ces bits de parité permettent de s'assurer que les données écrites dans le journal restent inchangées. Chaque bloc (ligne) du journal comporte trois parties : la première partie contient une paire constituée des données et de l'adresse correspondante, la seconde partie est constituée d'une paire de bits `w` et `v`, et la troisième partie comprend la parité générée, comme indiqué dans la figure 5.6. Nous avons utilisé le code Hsiao $(41, 34)$ pour protéger les données stockées dans le journal. 7 bits de parité sont utilisés pour construire la matrice H_0 comme suit :

1. toutes les 1 des 7 combinaisons de colonnes de poids 1 sont utilisés ;
2. nous sélectionnons les 34 colonnes de poids 3 obtenues par toutes les combinaisons possibles de 3 parmi 7.

La matrice de contrôle de parité (H_0 pour le code Hsiao (41,34) est montrée dans la figure 5.4. Elle a les caractéristiques suivantes :

1. le nombre total de 1 dans la matrice H est égal à $7 + 3 \times 34 = 109$;
2. le nombre moyen de 1 dans chaque ligne est égal à $109 \div 7$.

De plus, ces codes sont codés et décodés de façon parallèle. Lors du codage, les bits du message sont chargés dans le circuit d'encodage en parallèle, et les bits de contrôle de parité sont générés simultanément. Lors du décodage, les bits reçus sont lus par le circuit de décodage. En parallèle, les bits du syndrome dont générés simultanément et les bits reçus sont corrigées. La détection d'erreur double est accomplie par l'examen du nombre de 1 dans les vecteurs du syndrome.

5.3 Architecture et fonctionnement du journal

L'espace interne de stockage du journal est divisé en deux parties, UVJ et UJ, comme indiqué dans la figure 5.5. À la fin de chaque séquence valide, les données non encore validées UVD sont converties en données validées VD, et la ligne virtuelle qui sépare la partie haute de la partie basse du journal est décalée vers le haut pour tenir compte de la nouvelle situation. Les anciennes données validées VD sont transférées à la mémoire sûre au cours de l'exécution de la séquence actuelle.

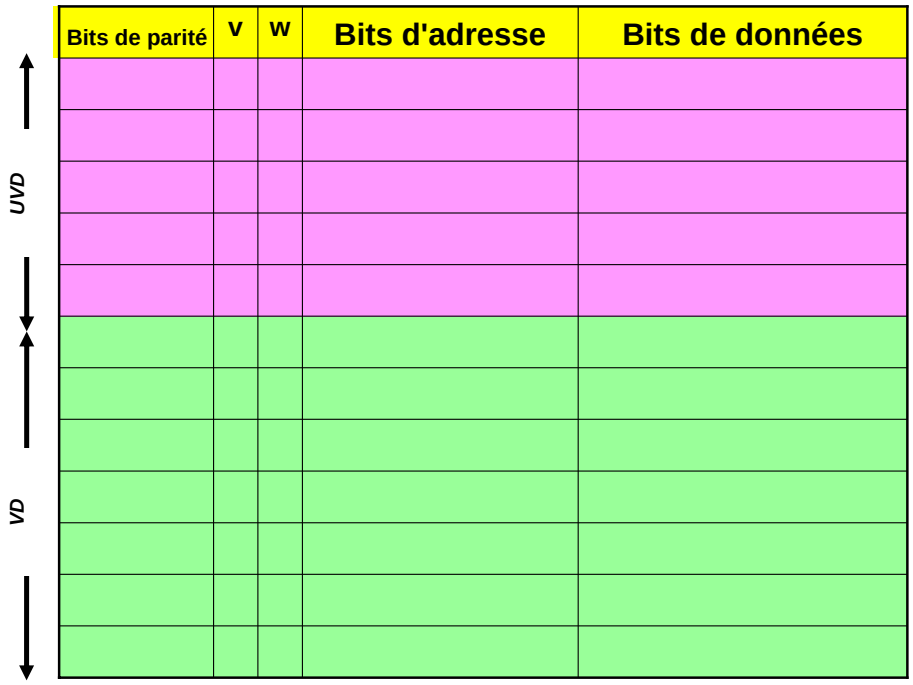


FIGURE 5.5 – Structure du journal

Comme expliqué dans la section 5.1, nous avons sélectionné le code Hsiao (41,34), une approche systématique pour la SEC-DED [Hsi10]. Chaque ligne du journal contient 41 bits. Les bits v et w bits serseront présentés plus loin. Associés au 16 bits d'adresse et au 16 bits de données, ils constituent les informations correspondant à un bloc du journal. Les bits restants sont les bits de contrôle de parité et représentent l'information de redondance liée au code correcteur d'erreur et protègent les autres bits, comme indiqué dans la figure 5.6.

Le système repose sur le modèle 2 présenté dans le chapitre 3, dans lequel les données ne peuvent pas être écrites directement dans la mémoire principale (cf. figure 5.7), afin de s'assurer que son contenu soir toujours digne de confiance. Les données sont d'abord écrites dans le journal, puis seulement en mémoire. L'adresse correspondante est toujours recherchée dans la partie non validée du journal, ainsi il ne peut y avoir deux données correspondant à la même adresse dans cette zone. Si l'adresse est trouvée, la donnée est mise à jour. Sinon, une nouvelle ligne est générée dans la partie non validée avec $w = 1$ et $v = 0$; les champs adresse, donnée, et bits de parité sont remplis avec les valeurs adéquates. Les bits w et v sont utilisé pour désigner les données modifiées et validées, respectivement.

Avant leur transfert à la mémoire, les données attendent la validation de la séquence actuelle. Le délai d'attente dépend du nombre d'instructions en cours d'exécution dans une séquence, et donc de la valeur de SD (*sequence duration*). Si aucune erreur n'est trouvée à la fin de la séquence actuelle, le processeur valide la séquence. Tous les données non encore validées du journal sont alors validées en positionnant le bit v correspondant à 1. Sinon, si une erreur est détectée, la séquence n'est pas validée et les données non encore validées sont révélées en positionnant le bit w correspondant à 0. Seules les

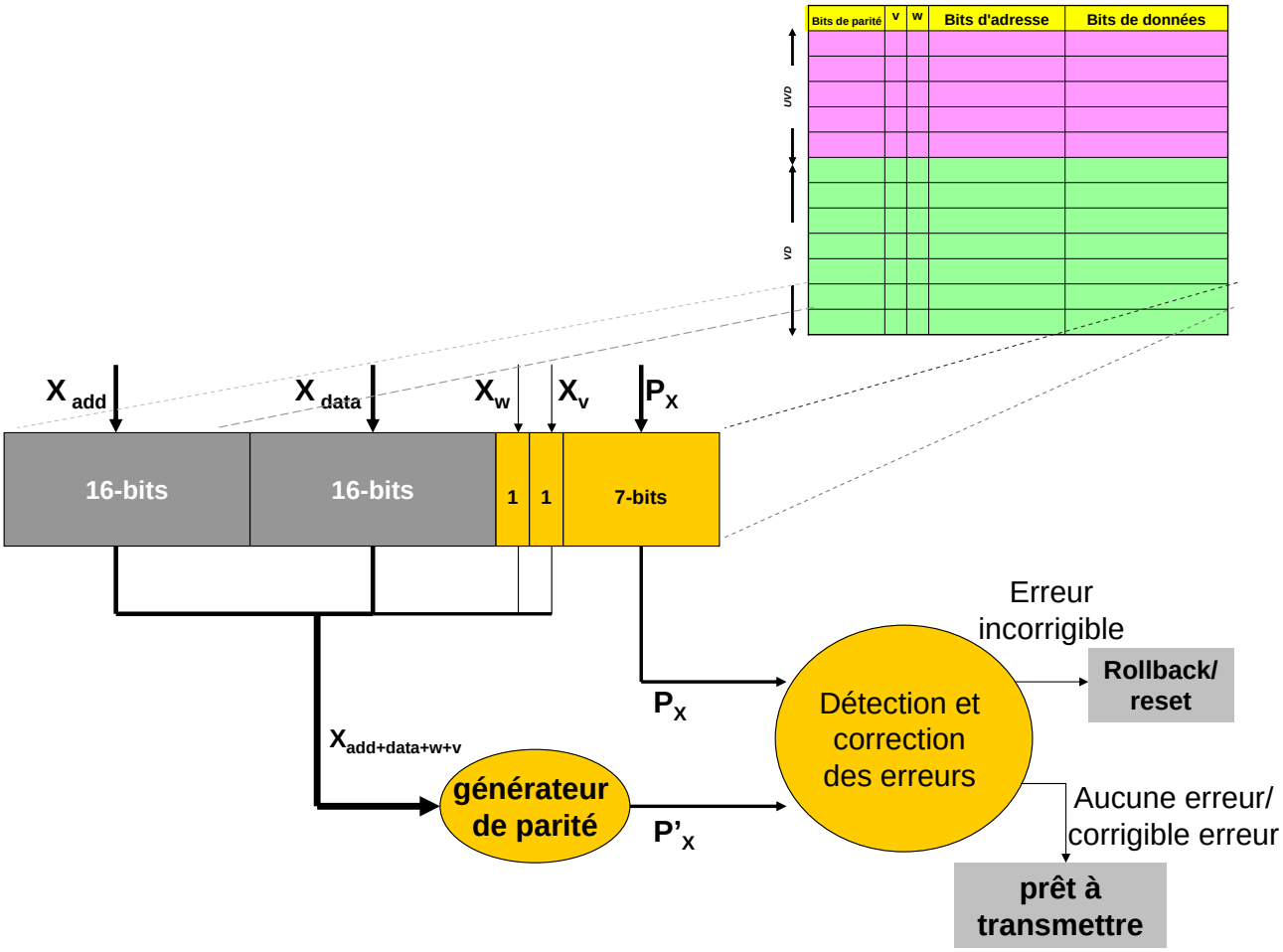


FIGURE 5.6 – Détection et correction des erreurs dans le journal : schéma de principe d'un bloc mémoire.

données ayant $v = 1$ peuvent être transférées à la mémoire sûre.

Notons que les dernières instructions d'une séquence sont utilisées pour écrire le contenu des éléments sensibles dans le journal. Lors de la validation de la séquence, ces données voient leur bit v positionné à 1, et par conséquent sont prêtes à être stockées en mémoire principale. En cas de non-validation de la séquence (cf. figure 5.8), les éléments sensibles sont restaurés à partir de la mémoire lors du *rollback* alors que les données non validées du journal sont rejetées, et l'exécution est relancée à partir du point de validation VP précédent. Chaque ligne du journal est protégée par un code Hsiao comme indiqué dans la figure 5.6. Cette protection est mise en œuvre de la façon suivante :

- une erreur détectée dans la zone de données non validées entraînera l'invalidation de la séquence (*rollback*) ;
- la zone de données validées du journal VD est écrite ligne par ligne en mémoire principale. La zone VD étant une copie de la dernière séquence valide, la perte de ces données empêcherait l'achèvement de l'exécution du programme ou du thread et nécessiterait une réinitialisation complète du système. Cela ne peut survenir que si une erreur est détectée qui surpasse la capa-

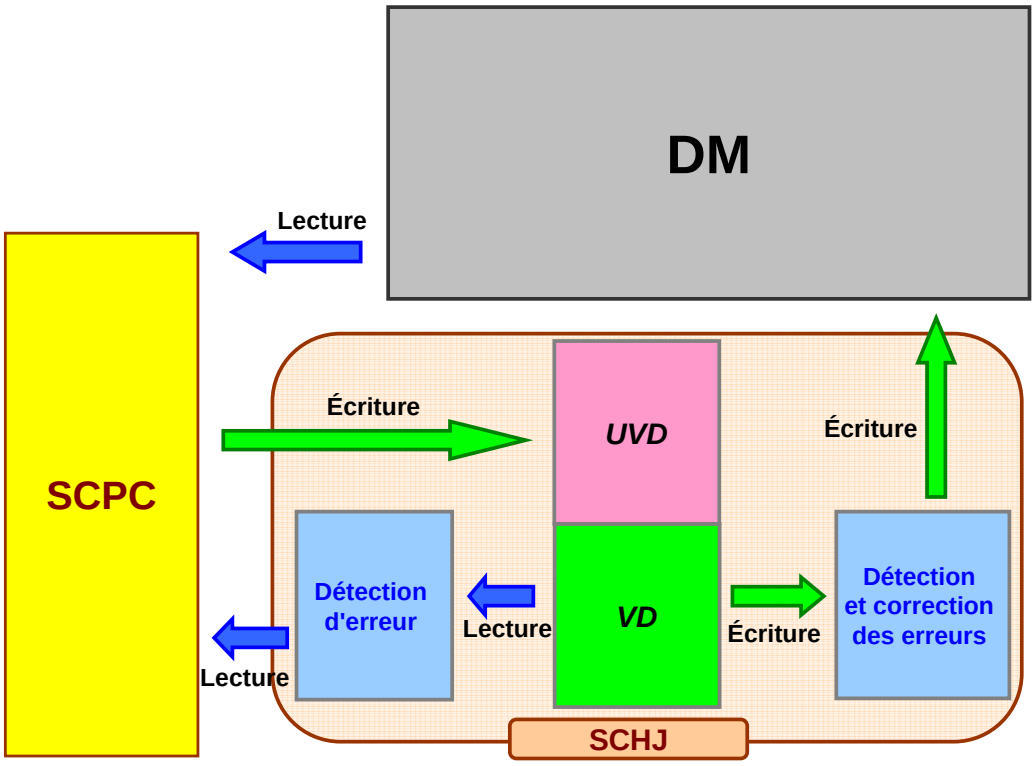


FIGURE 5.7 – Architecture globale

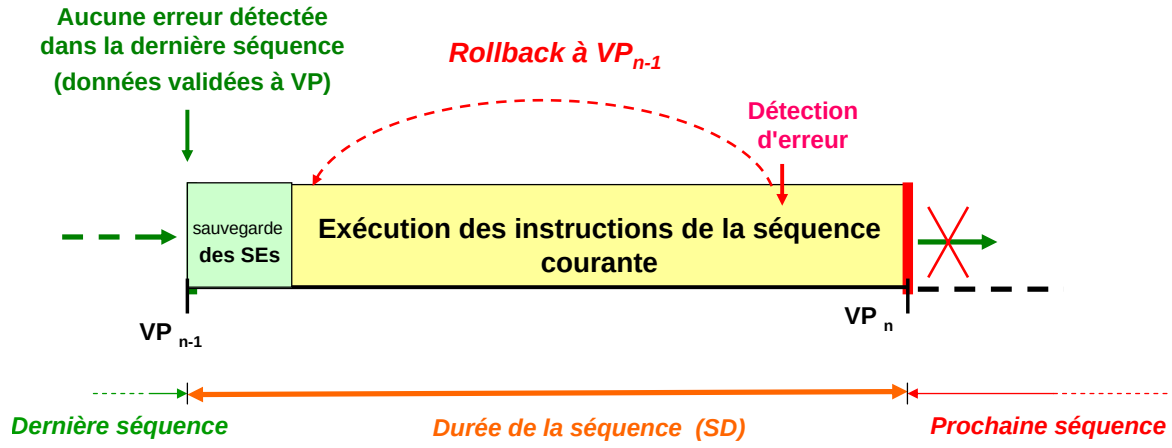


FIGURE 5.8 – Mécanisme de *rollback* après détection d'erreur

cité de correction du code (par exemple, deux bits erronés sur la même ligne de la partie validée du journal).

Des explications supplémentaires sur le fonctionnement du *rollback* peuvent être trouvées dans [RAM+09, AMD+10, ARM+11].

Comme indiqué précédemment, la mémoire sûre est intégrée à la puce, et donc supposée être suffisamment rapide pour remplir les exigences de performance de notre processeur. Notre stratégie consistant à utiliser un journal a pour but non seulement d'améliorer la tolérance aux fautes, mais

aussi de permettre au mécanisme de *rollback* d'être mis en œuvre avec une faible pénalité temporelle comparé à une approche totalement matérielle ou pas de protection du tout.

5.3.1 Modes de fonctionnement du journal

Le fonctionnement du journal est représenté dans l'organigramme de la figure 5.9. Quatre modes de fonctionnement sont résumés dans la table 5.1. Le circuit vérificateur ECC est activé lors de chaque accès en lecture et en écriture à la mémoire. Les feux de circulation des figures 5.10, 5.11, 5.13 et 5.15 représentent des flux de données relatifs à l'opération d'écriture, car lors des opérations de lecture le journal est totalement transparent pour le processeur.

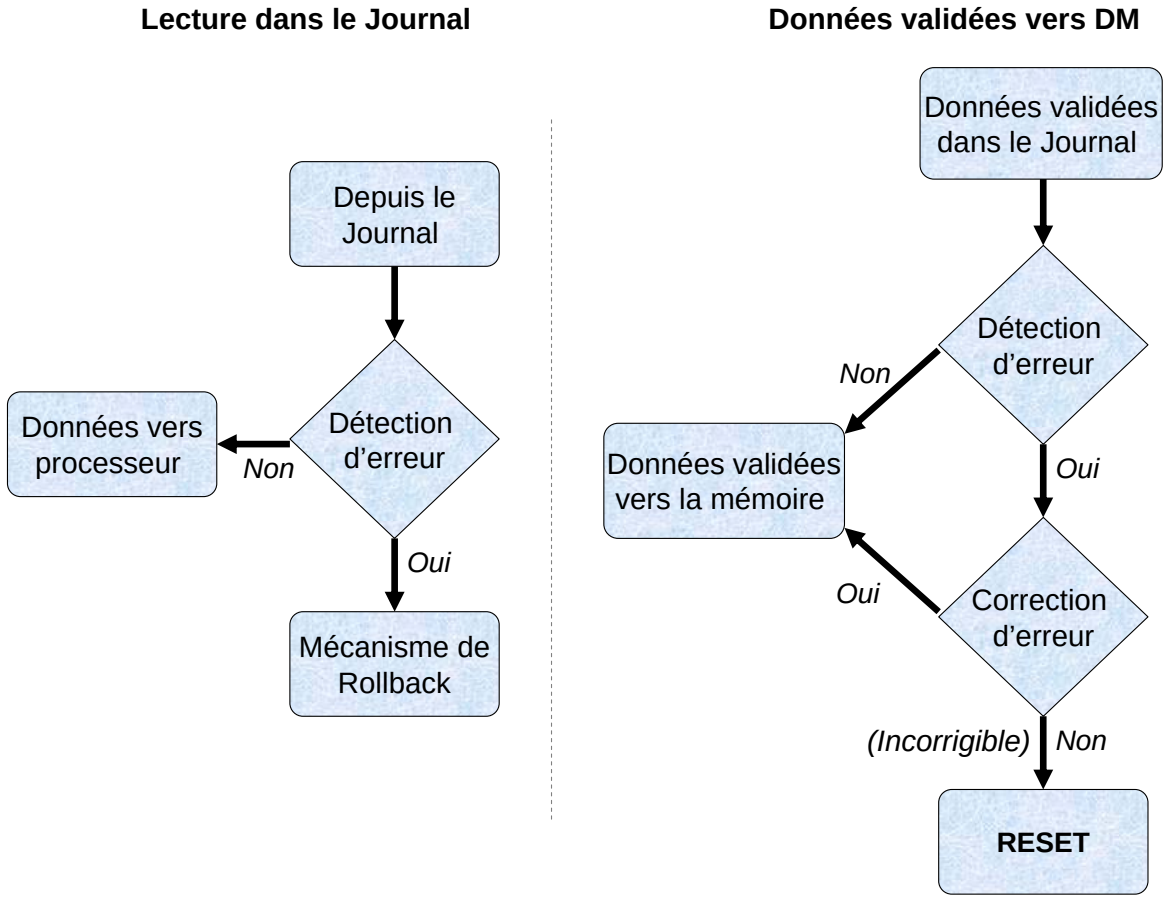


FIGURE 5.9 – Organigramme de fonctionnement du journal

TABLE 5.1 – Modes des Journal

Mode	Opération
00	initialisé
01	lecture/écriture
10	valide (v= 1)
11	non valide (*rollback*)

Mode 00 – Ce mode est actif au démarrage du programme ou lors du redémarrage si une erreur non corrigible est détectée dans les données validées du journal. Dans ce mode, le processeur est réinitialisé et ré-exécute le programme à partir des valeurs par défaut, en annulant toutes les données stockées dans le journal. Tous les bits w et v sont forcés à 0 ($v \leftarrow 0, w \leftarrow 0$). Il n'y a aucun échange de données entre le processeur, le journal ou la mémoire principale, comme le montre la figure 5.10.

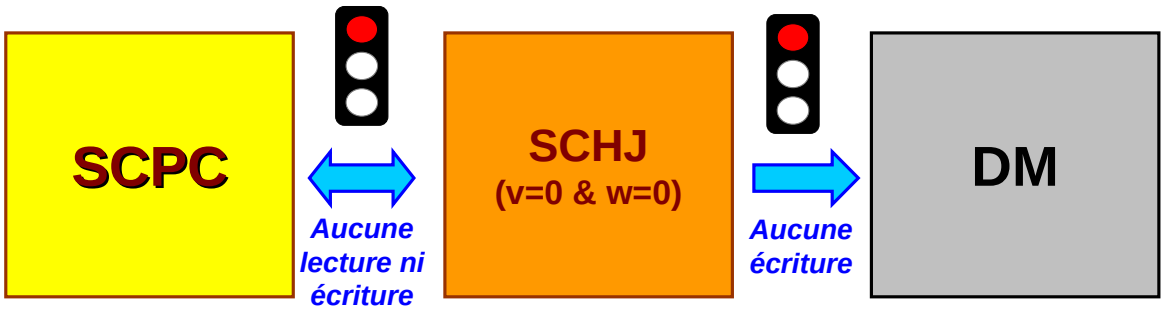


FIGURE 5.10 – Mode 00 du journal

Mode 01 – C'est le mode de fonctionnement normal en lecture ou en écriture selon l'instruction active dans le journal ($rd = 1$ ou $wr = 1$). Dans ce mode, le processeur peut écrire directement dans le journal, mais pas dans la mémoire, afin d'éviter le risque de contamination de celle-ci par les données. Cependant, il a accès en lecture à la fois au journal et à la mémoire principale (non représenté sur la figure 5.11 pour éviter la complexité). Les données lues à partir du journal sont vérifiées afin de détecter les possibles erreurs. En cas de détection d'erreur, le processeur passe en mode 11 dans lequel mécanisme de restauration est activé sans attendre le point de la validation de la séquence actuelle.

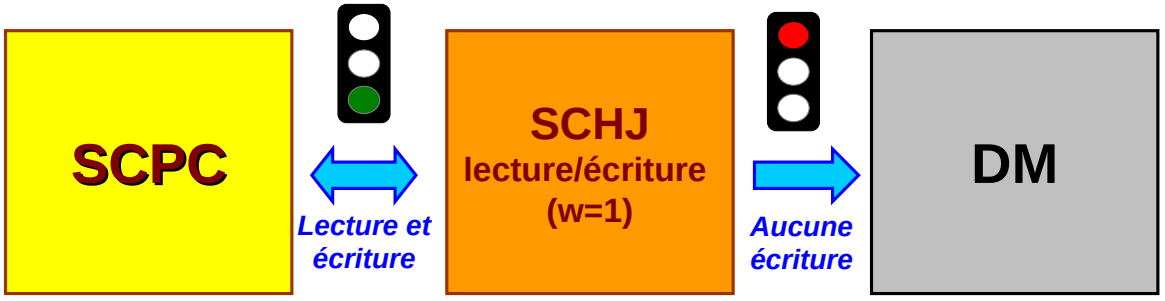


FIGURE 5.11 – Mode 01 du journal

Dans des conditions normales de fonctionnement, le processeur est principalement en mode 01. Comme indiqué dans la figure 5.12, lorsque le processeur a besoin de lire des données dans le journal, il cherche parmi les tags d'adresse celui qui correspond aux données cherchées (représenté par la flèche a sur la figure 5.12). Si l'adresse demandée est trouvée, avant leur transfert vers le cœur de processeur, les données correspondantes sont vérifiées par comparaison des bits de parité stockés avec les bits de parité générés par l'unité de détection d'erreur, comme indiqué sur la figure 5.12.

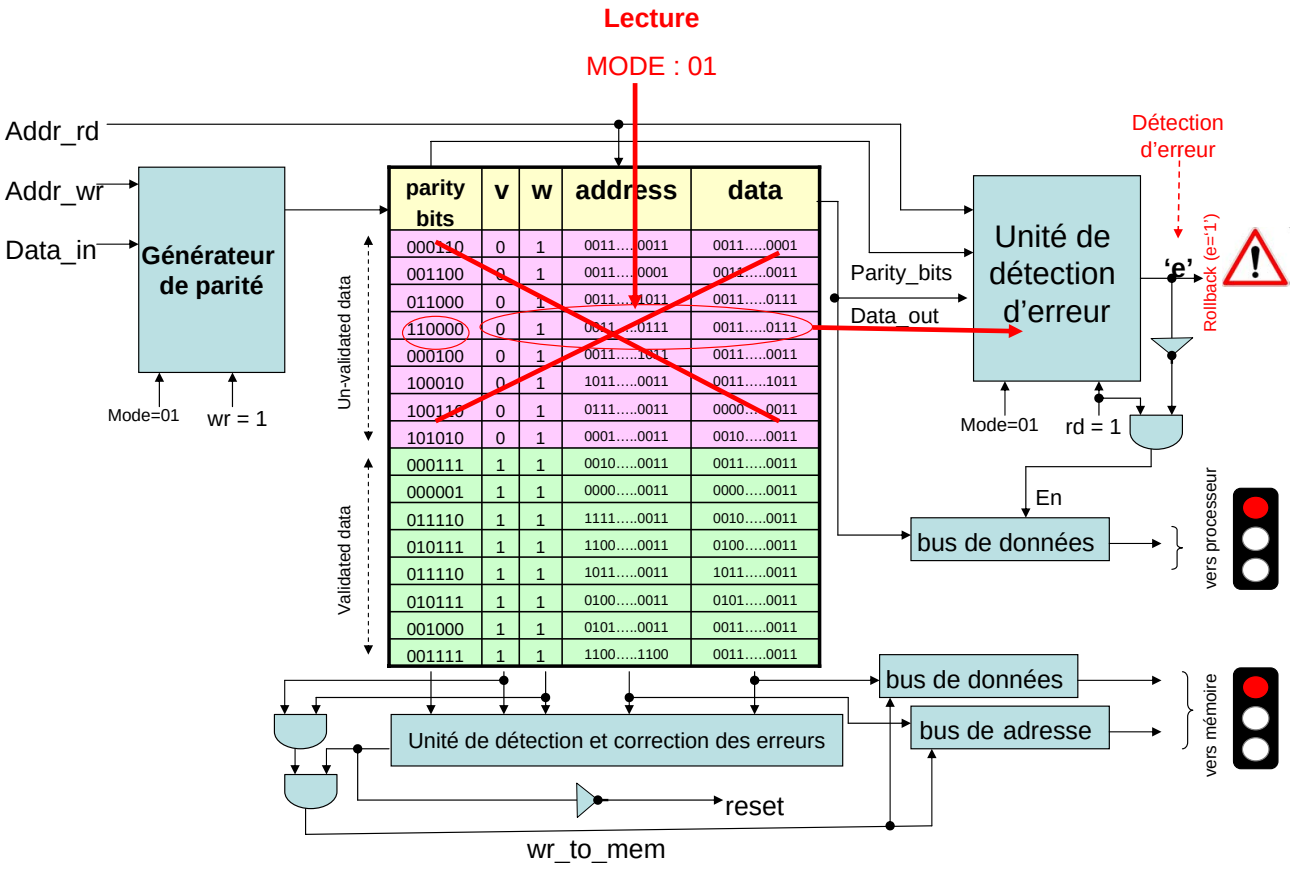


FIGURE 5.12 – Lecture des UVD dands le journal en mode 01

- si une erreur est détectée (cf. figure 5.12), le mécanisme de restauration est invoqué car la partie non validée du journal contient les données générées lors de la séquence actuelle (indiquées par le champ v positionné à 0). Le signal *En* (*enable*) sur le bus de données est alors positionné à 0 pour interdire tout autre transfert de données du journal vers le cœur. Toutes les données écrites pendant cette séquence sont ignorées ($w \leftarrow 0$).

- si aucune erreur n'est détectée, les données demandées sont envoyées au cœur. Le bus de données entre le journal et le cœur du processeur est activé alors que le bus de données entre le journal et la mémoire principale est temporairement désactivé. Notez dans la figure 5.12 : la flèche 1 montre $w = 1$, ce qui indique les données écrites dans le journal, tandis que la flèche 2 montre les données validées lors de la séquence précédente.

Mode 10 – ce mode concerne le transfert des données validées vers la mémoire (voir figure 5.13). Ce mode est activé lors du point de validation si aucune erreur n'a été détectée lors de la séquence actuelle (c'est-à-dire la séquence en cours de validation), et par conséquent les données écrites au cours de cette séquence voient leur bit v positionné à 1 ($v \leftarrow 1$), ou, dans le pire des cas, avant le point de validation si le journal est complètement rempli. Ensuite, les données fraichement validées peuvent être envoyées à la mémoire principale en fonction de la disponibilité du bus de données.

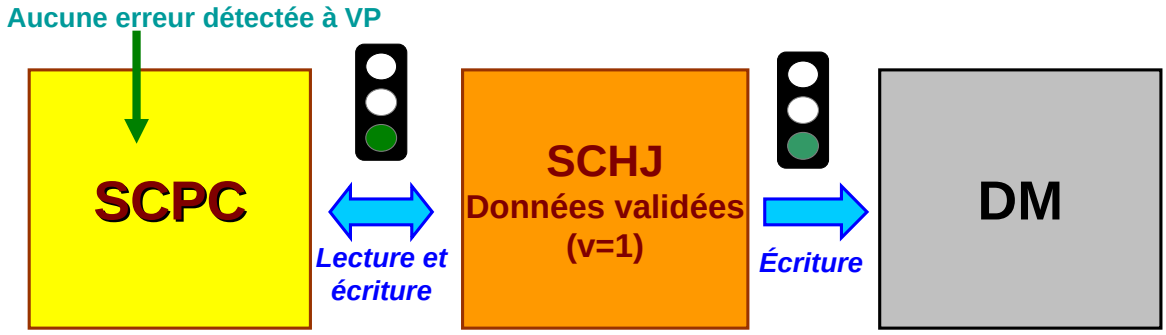


FIGURE 5.13 – Mode 10 du journal

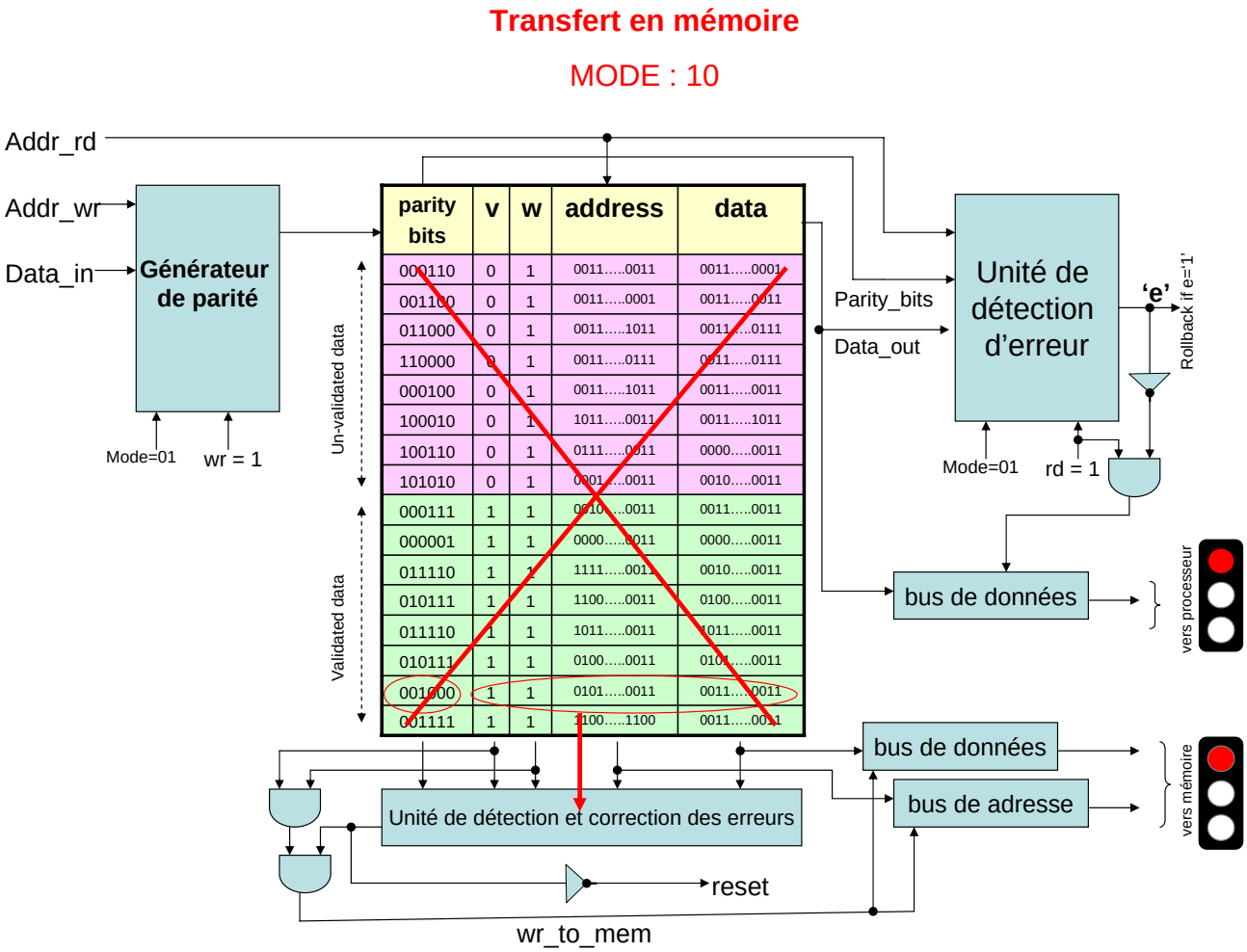


FIGURE 5.14 – Mode 10 du journal en cas d'erreur non-corrigibledétectée

Alors que les données validées sont en cours d'écriture en mémoire, elles sont vérifiées pour détecter les erreurs *soft* éventuelles qui peuvent les avoir corrompues lors de leur séjour dans le journal (voir figure 5.3 (b)).

- si une erreur corrigible est détectée, alors les données sont envoyées à la mémoire après correction ;

- si une erreur non corrigible est détectée (la probabilité pour que ceci arrive est cependant très faible [Gha11]), par exemple une double erreur survenant dans le même cycle d'horloge et affectant la même ligne de la mémoire (voir la flèche b dans la figure 5.14), alors une situation non récupérable se pose car il n'y a aucun moyen de revenir à un précédent point de validation (les données correspondantes ne sont plus disponibles en mémoire principale, puisqu'elles ont été copiées à partir du journal). La réinitialisation du cœur, en forçant le mode 00 (et éventuellement en positionnant un indicateur d'alarme), est le comportement habituel dans ce type de situation.

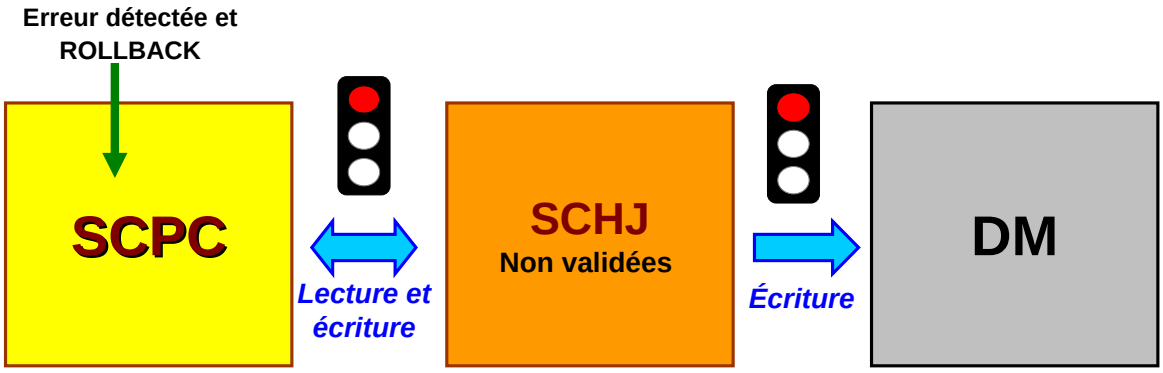


FIGURE 5.15 – Mode 11 du journal

Mode 11 – ce mode, qui a été partiellement présenté avec le mode 01, est invoqué quand une erreur est détectée dans la partie non validée du journal (cf. figure 5.3 (a)) pendant une opération de lecture/écriture comme le montre la figure 5.15. Dans ce mode, toutes les données écrites dans la partie non validée du journal (c'est-à-dire l'ensemble des données générées lors de la séquence en cours) sont invalidées et rejetées ($w \leftarrow 0$) comme indiqué par la croix dans la figure 5.12. Le *rollback* est invoqué pour relancer l'exécution de la séquence actuelle depuis le début, après restauration des dernières valeurs sûres des éléments sensibles. Après succès du processus de restauration, le mode 01 (mode lecture/écriture) est réactivé.

5.4 Risque de contamination des données

À l'intérieur de la partie non validée du journal, nous pouvons détecter et recouvrer jusqu'à deux erreurs binaires en nous appuyant sur le code de Hsiao pour la détection, et sur le mécanisme du *rollback* pour le recouvrement. La pénalité temporelle maximale pour corriger l'erreur sera égale à la longueur de la séquence, en cas de *rollback*.

D'autre part, à l'intérieur de la partie validée du journal, nous pouvons uniquement corriger les erreurs simples (SBU). Si un MBU affectant deux bits est détecté alors le programme devra être réexécuté, ce qui peut entraîner des contraintes fortes sur les performances en temps réel, mais notre

hypothèse sur la mémoire principale sûre reste valide. Par ailleurs, la probabilité d'occurrence d'un MBU étant nettement moindre que celle d'un SBU [QGK+06], il y a de très faibles chances qu'une telle situation se produise.

Cela signifie que – du point de vue de la fiabilité – la partie validée du journal est une zone de stockage de données bien plus critique que la partie non validée. Par conséquent, il est important de savoir combien de temps les données restent à l'intérieur de la zone validée. En fait, toutes les données stockées dans cette partie du journal sont transférées à la mémoire au cours d'une séquence. Cela signifie que la durée maximale du risque de contamination des données est égale à la longueur de la séquence (SD). Plus SD est grand, plus le risque de contamination des données à l'intérieur du journal se pose.

5.5 Résultats d'implantation

Le journal a été modélisé en VHDL au niveau RTL, et mis en œuvre sur une carte Altera Stratix III EP3SE50F484C2 à l'aide de Quartus II. Les résultats obtenus en terme de surface pour une profondeur de journal égale à 10 sont rapportés dans la table 5.2. À partir de ces résultats, les observations suivantes peuvent être faites :

- selon sa profondeur, le journal occupe environ 40 à 50 % de la surface totale ;

TABLE 5.2 – Implémentation

	ALUTS comb.	Logique dédiée
SCHJ	591	399
SCPC et SCHJ	1452	677

Dans le cas où une erreur non corrigible (par exemple une double erreur dans une seule ligne) est détectée dans la partie validée du journal (VJ), alors même par *rollback* nous ne pouvons pas recouvrer cette erreur car les données n'appartiennent pas à la séquence en cours ; dans ce cas, le processeur doit être réinitialisé comme indiqué dans la figure 5.16.

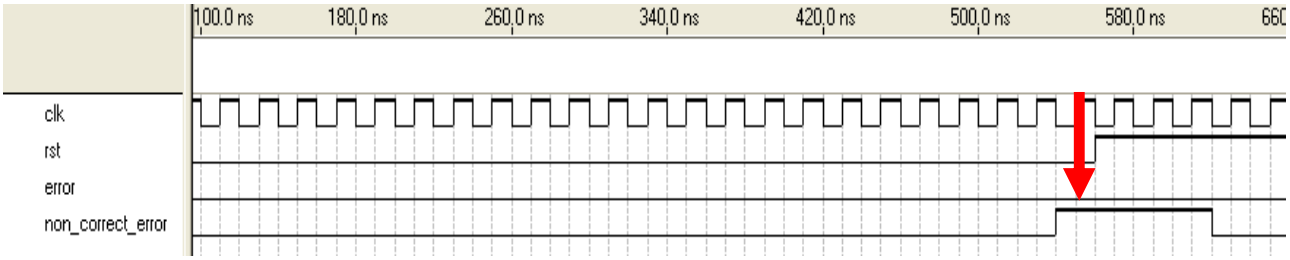


FIGURE 5.16 – Détection d'une erreur non corrigible

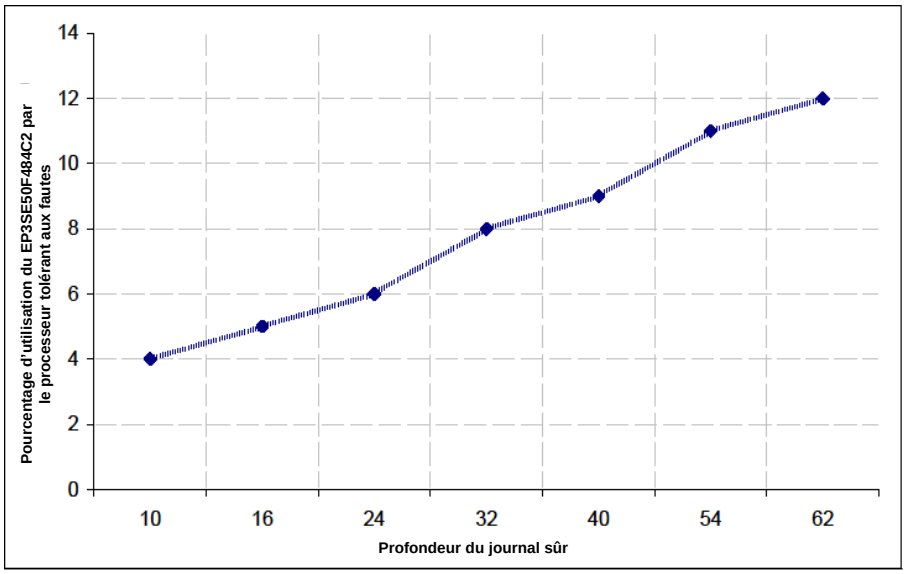


FIGURE 5.17 – Augmentation du pourcentage d'utilisation des ressources de l'EP3SE50F484C2 par le processeur tolérance aux fautes (SCPC + SCHJ) en fonction de l'augmentation de la profondeur

5.5.1 Minimisation de la taille du journal

À partir des résultats d'implantation présentés dans le tableau 5.2, nous constatons que le journal occupe un pourcentage important de la superficie totale du processeur tolérance aux fautes. Nous avons étudié l'impact sur le pourcentage d'utilisation du processeur par rapport à la profondeur du journal. Les résultats sont rapportés dans la figure 5.17. Ils montrent que le surcoût matériel dépend directement de la profondeur du journal.

En fait, la profondeur du journal est un paramètre relatif et dépend du type de benchmark employé et de la durée de la séquence. Du point de vue théorique, la partie non validée du journal devrait être égale à la longueur maximale de la séquence si le programme exécuté par le processeur ne comprenait que des instructions qui nécessitent d'écrire dans la mémoire (par exemple, une série d'instructions `DUP` (duplication) comme sur la figure 5.18). À chaque exécution de l'instruction, le contenu de NOS est écrit dans la mémoire. La taille requise pour la partie non validée du journal devrait donc être égale à la longueur de la séquence (voir sur la figure 5.19 la flèche *a*). D'autre part, le cas extrême le plus favorable est possible pour les programmes contenant des instructions qui n'ont pas ou très peu besoin d'écrire dans la mémoire (comme la série de `SWAP` de la figure 5.18). La profondeur requise pour le journal est alors minimale.

Cependant, pour répondre à de réelles applications industrielles, il est nécessaire de trouver des relations entre la profondeur du journal et la longueur de la séquence. En conséquence, nous avons calculé le pourcentage des instructions d'écriture dans les benchmarks déjà présentés (voir la section 3.6). Ils sont coûteux en trafic processeur-mémoire parce qu'ils lisent et écrivent toujours les données de ou vers la mémoire. Or le résultat montre que le pourcentage maximum d'écriture en mémoire est de 39 % (voir la figure 5.19, flèche *b*). Cependant, nous ignorons les écritures aux mêmes adresses

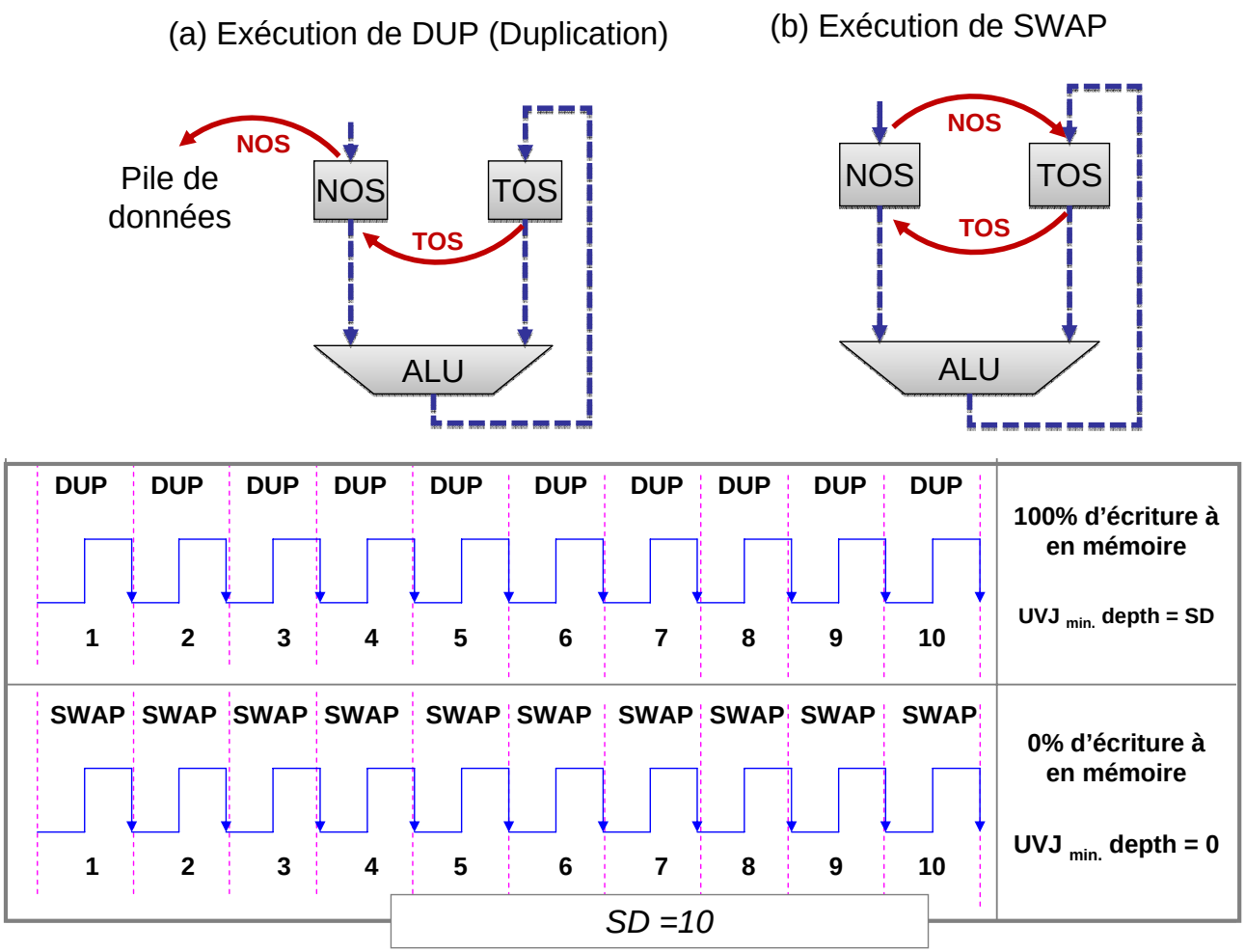
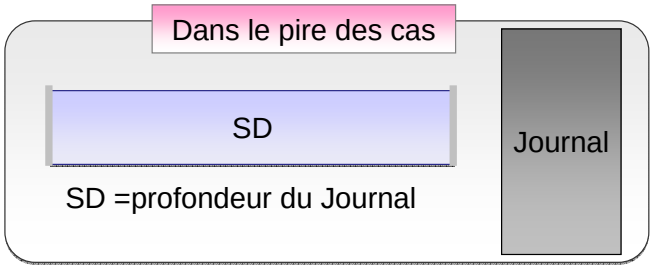


FIGURE 5.18 – Limites théoriques de la profondeur du journal

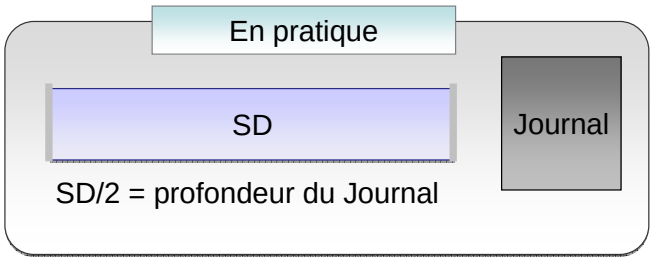
mémoire qui permettent de réduire encore la profondeur requise pour le journal. Cela montre que la profondeur pratique du journal ne devrait pas excéder 50 % de la longueur de la séquence (compte tenu de onze pourcent de marge de sécurité). Ici, il faut noter que les résultats d'occupation en surface précédemment présentés ont été calculés dans le pire des cas (donc lorsque UVJ = SD). Toutefois, la profondeur nécessaire pour le journal est de SD/2.

Maintenant, pour finaliser le calcul de la profondeur optimale du journal, il est important de trouver la relation entre la durée de la séquence et la dégradation des performances. En conséquence, nous avons développé un modèle de processeur en utilisant des outils C++ dédiés. Les erreurs sont injectées artificiellement dans le modèle de processeur simulé. Ici, seules des injections de SBU ont été considérées. L'environnement expérimental complet sera présenté dans le chapitre 6. Le facteur CPO (*clock per operation*, ou nombre de cycles d'horloge par opération) est choisi pour déterminer la dégradation des performances due à la réexécution. Idéalement le processeur exécute une instruction par cycle d'horloge, ce qui signifie que CPO $\approx$ 1. La discussion de la section 3.1.2 montre que dans un système BER, la dégradation des performances repose sur deux facteurs : le taux de réexécution (*rollback*) et le ratio d'exécution des instructions effectif, (SD-SED)/SD. Plus la dégradation des



Point de vue théorique	
Benchmark	% d'écriture
Séries de DUP	100
Séries de Swap	0

(a)



Point de vue pratique	
Benchmark	% d'écriture
Tri à bulle	38
Multiplication de matrice	39
Contrôle	36

(b)

FIGURE 5.19 – Relation entre la profondeur journal et le pourcentage d'instructions d'écritures dans le programme

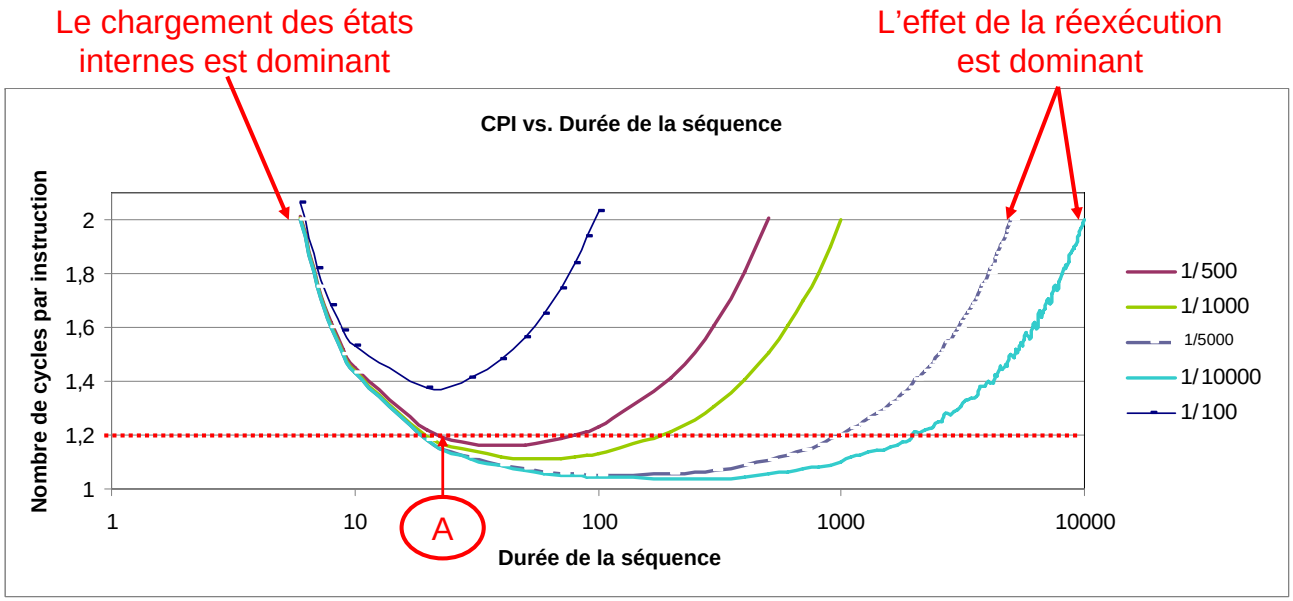


FIGURE 5.20 – CPI vs. SD

performances sera importante, plus le CPO sera élevé.

Les courbes CPO vs. SD ont été établies (cf. figure 5.20) pour différents taux d'injection d'erreurs. Les courbes obtenues suivent une forme en « U », car pour des valeurs faibles de SD le temps de chargement des états internes domine, et pour valeurs élevées de SD le temps dû à la réexécution des instructions est dominant. Ces courbes montrent que des profondeurs importantes de journal sont possible uniquement avec des taux d'injection d'erreur faibles. Ainsi, pour un taux donné, il existe

des limites à l'intérieur desquelles le processeur continue d'offrir de bonnes performances.

Si nous acceptons une dégradation de performances de 20 %, alors la durée de séquence minimale s'avère être de 20 (voir la flèche *A*). En bref, la profondeur de journal pratique se situe quelque part autour de 10 si nous acceptons une profondeur de journal égale à 50 % de la longueur de la séquence.

5.5.2 Durée de séquence dynamique

Dans le modèle présenté, nous avons utilisé une durée de séquence fixe qui présente un surcoût à la fois en surface et en performance. Cependant, avec une durée de séquence dynamique, ces problèmes peuvent être résolus. La figure 5.21 montre une valeur moyenne pour SD. Cela peut nous permettre d'utiliser une séquence plus longue avec une profondeur de journal plus faible. Cela peut améliorer le surcoût en surface et la dégradation de performances pour des taux d'injection d'erreurs faibles.

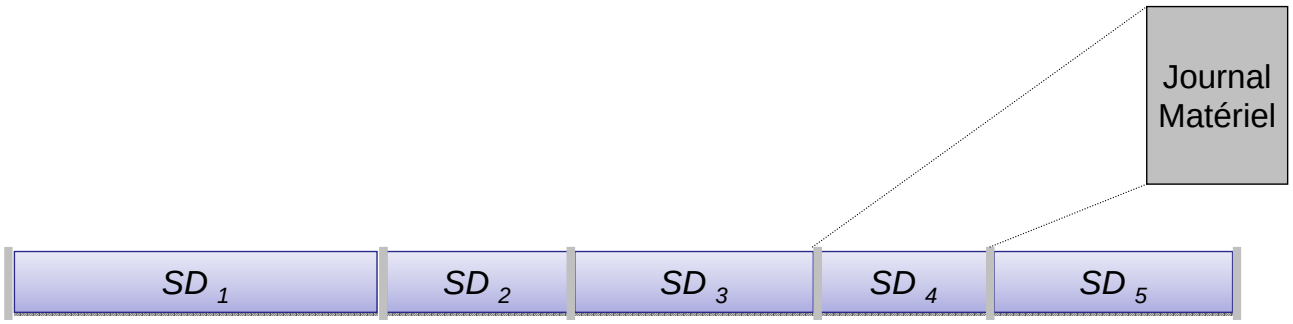


FIGURE 5.21 – Durée de séquence dynamique

Par ailleurs, cela peut permettre de reconfigurer dynamiquement la durée de la séquence en fonction du taux d'erreur. Par exemple, si la séquence est invalidée à plusieurs reprises, alors le système va automatiquement réduire sa durée pour l'ajuster en fonction du taux d'erreur. L'inconvénient est que cela peut augmenter la complexité de gestion du journal. Une durée de séquence dynamique est une considération importante à prendre en compte dans le futur pour réduire le coût matériel de la protection.

5.6 Conclusion

La présence du journal facilite le mécanisme de *rollback* d'une part, et il permet de masquer les erreurs – SBU et MBU de 2 bits – et interdire leur propagation à la mémoire principale d'autre part. Pour réduire le coût matériel, le code de Hsiao a été utilisé. Il offre une détection efficace des erreurs doubles et la correction des erreurs simples. Grâce à un accès parallèle à la mémoire et au journal simultanément dans des opérations de lecture, l'efficacité globale du système a été augmentée.

Le journal occupe un pourcentage important de la surface du processeur tolérant aux fautes. Réduire la taille du journal peut efficacement réduire la surface globale. La taille du journal dépend du

type de programme devant être exécuté par le processeur. Pour des applications pratiques, sa profondeur peut être égale à la moitié de la durée de la séquence, mais une réduction supplémentaire est possible en mettant en œuvre des longueurs de séquences dynamiques plutôt que fixes.

Dans le prochain chapitre, nous étudions le taux de couverture d'erreurs et la dégradation des performances due à la réexécution.

Chapitre 6

Validation du processeur tolérant aux fautes

Dans les chapitres précédents, nous avons conçu un processeur tolérant aux fautes basé sur une capacité de détection d'erreurs concurrente et sur une stratégie de recouvrement d'erreur par *rollback*. Le design tolérant aux fautes est construit sur un cœur de processeur autocontrôlé dont l'architecture suit la philosophie MISC et sur un journal matériel autocontrôlé qui empêche les erreurs de se propager vers la mémoire et limite l'impact du mécanisme de *rollback* sur les performances temporelles. Les architectures du processeur et du journal matériel autocontrôlés ont été discutées dans les chapitres 4 et 5, respectivement.

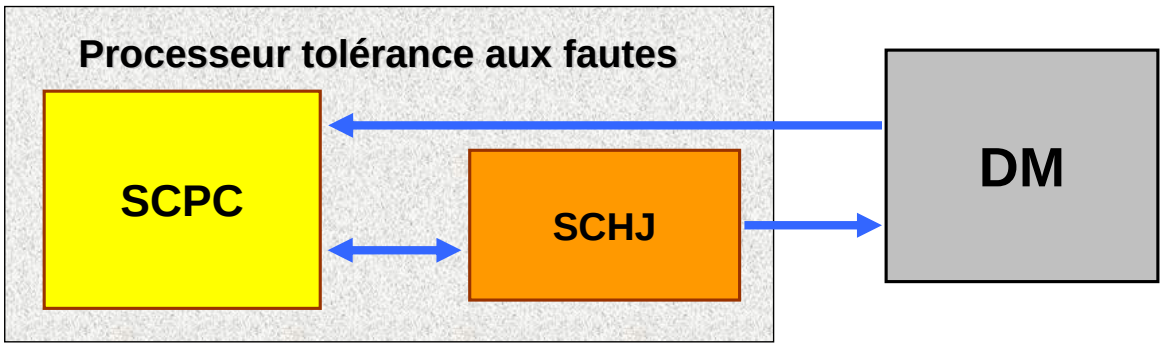


FIGURE 6.1 – L'architecture de processeur tolérant aux fautes devant être validée

Dans ce chapitre, nous allons évaluer la capacité de tolérance aux fautes du processeur – cœur (SCPC) + journal (SCHJ) – pour valider notre stratégie de conception, comme indiqué dans la figure 6.1. L'évaluation sera effectuée par simulation. L'injection contrôlée d'erreur sera utilisée pour forcer artificiellement le processeur à affronter des situations anormales. La capacité de tolérance aux fautes du processeur sera jugée par le calcul du ratio des erreurs détectées sur les erreurs injectées pour différents scénarios de simulation, c'est-à-dire pour différentes applications de référence (benchmarks) et différents profils d'injection d'erreur. La performance temporelle sera également évaluée.

Ce chapitre est organisé comme suit. D'abord nous allons analyser l'hypothèse de conception qui a été établie dans la méthodologie et par conséquent, les propriétés du processeur tolérant aux fautes à vérifier. Ensuite, après une courte présentation de la méthodologie d'injection d'erreur, les résultats

expérimentaux sont présentés et discutés, du point de vue de la tolérance aux fautes ainsi que des performances temporelles. Enfin, nous comparerons la méthode proposée avec la méthodologie de conception du processeur LEON 3 FT.

6.1 Hypothèses de conception et propriétés à vérifier

Dans le cœur, les codes de parité et les codes à résidu sont employés pour détecter les erreurs dans les registres internes et les circuits arithmétiques et logiques de l'ALU. Selon les hypothèses, la mémoire principale est un lieu sûr où les données demeurent non corrompues. Par conséquent, les données à risque sont interdites d'accès à la mémoire. Ceci est réalisé par le journal qui intègre un mécanisme de détection d'erreurs ainsi qu'une certaine capacité de correction d'erreur. Son rôle est de simplifier la gestion des données validées et non validées et d'accélérer le mécanisme de *rollback* utilisé pour le recouverement d'erreur.

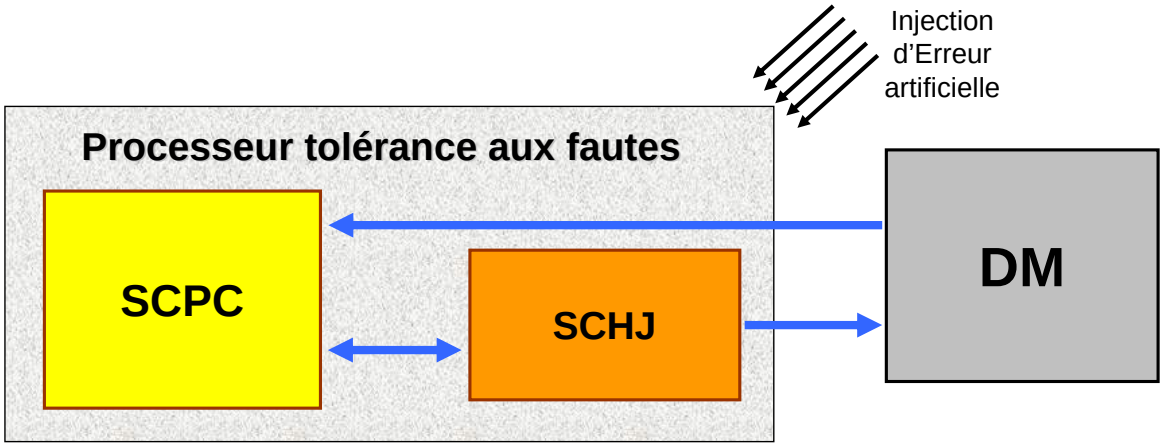


FIGURE 6.2 – Injection d'erreurs dans le processeur tolérant aux fautes

La capacité de tolérance aux fautes du processeur doit être évaluée en tant que la capacité de gestion correcte des erreurs qui apparaissent dans n'importe quelle partie du cœur ou du journal (figure 6.2), avec différents profils d'erreurs à tester, c'est-à-dire en utilisant des motifs différents et des taux d'erreurs variables. La dégradation de performance en vitesse sera également évaluée en même temps que la capacité de tolérance aux fautes, puisque l'impact du *rollback* augmentera proportionnellement, du fait de l'augmentation du taux de réexécution. En conséquence, le taux de *rollback* par rapport au taux d'injection d'erreurs peut également être évalué.

En résumé, nous allons étudier la fiabilité et les performances globales de l'architecture proposée pour le processeur tolérant aux fautes en relevant les défis suivants dans les sections à venir :

- efficacité de l'autocontrôle du processeur ;
- dégradation des performances due à la réexécution ; et
- effet de l'injection d'erreur sur le taux de *rollback*.

6.2 Méthodologie d'injection d'erreurs et profils d'erreurs

Avant d'aborder les défis mentionnés ci-dessus, il est nécessaire de choisir la méthodologie d'injection d'erreurs et les profils d'erreurs qui seront appliqués, c'est-à-dire les motifs d'erreurs et les taux d'erreur. L'injection de fautes dans le matériel d'un système peut être mise en œuvre de deux façons :

1. l'injection de fautes physique ;
2. l'injection de fautes simulée.

Dans ce travail, nous emploierons l'injection de fautes simulée – et plus précisément l'injection d'erreurs *soft* dues à des fautes transitoires – dans laquelle les erreurs sont injectées et altèrent les valeurs logiques lors de la simulation. L'injection basée sur la simulation est un cas particulier d'injection de fautes ou d'erreurs qui peut supporter différents niveaux d'abstraction du système tels que le niveau fonctionnel, architectural, logique ou électrique [CP02]. Pour cette raison, c'est une méthode largement utilisée pour étudier l'injection de fautes.

Par ailleurs, il existe plusieurs autres avantages à cette technique, le plus grand étant l'observabilité et la contrôlabilité de l'ensemble des composants modélisés. Un autre aspect positif de cette technique est la possibilité d'effectuer la validation du système pendant la phase de conception avant l'obtention d'un design final.

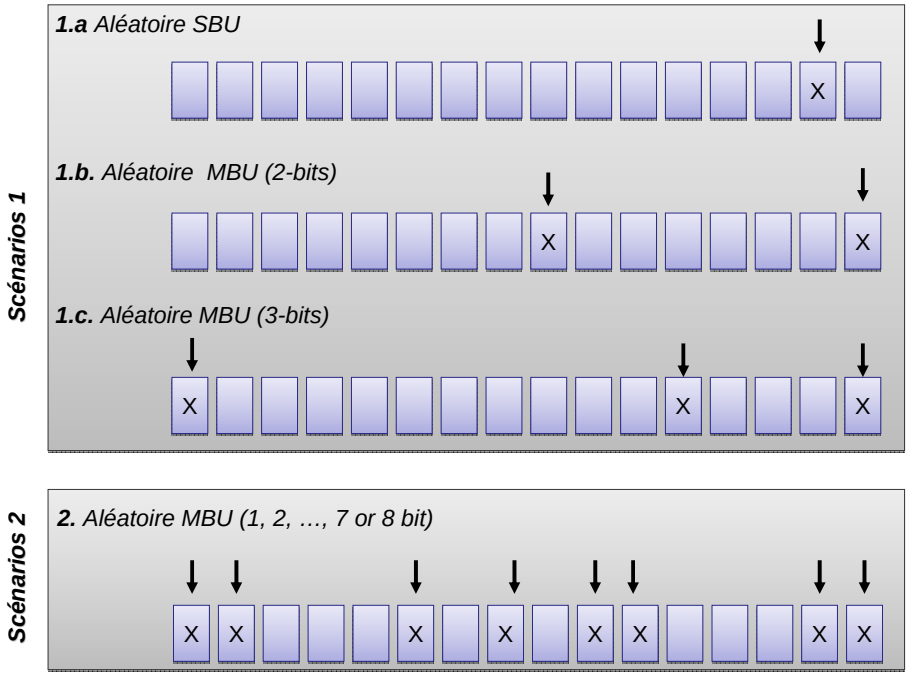


FIGURE 6.3 – Motifs d'erreurs (les erreurs peuvent survenir sur n'importe quel bit, pas nécessairement les bits présentés ici)

Les fautes considérées sont les SBU (changement d'un seul bit dans un seul registre) et les MBU (changements de plusieurs bits à la fois dans un seul registre). Ces modèles de fautes (SBU et MBU)

sont couramment utilisés avec les modèles RTL [Van08]. Les motifs d'erreurs exacts considérés dans ces expériences sont présentés dans la figure 6.3 : dans le scénario 1, nous avons considéré (a) une erreur aléatoire sur un bit, (b) une erreur aléatoire sur 2 bits, et (c) une erreur aléatoire sur 3 bits ; dans le scénario 2, les erreurs aléatoires sévères (de 1 bit jusqu'à 8 bits erronés dans un seul registre) sont considérées.

6.3 Validation expérimentale de la méthodologie d'autocontrôle

Nous allons évaluer la couverture d'erreur par l'injection de fautes simulées, l'objectif étant de déterminer l'efficacité de la méthode de tolérance aux fautes proposée, et donc de déterminer ses limites. Il est nécessaire de développer un environnement pour analyser les effets des fautes transitoires dans l'architecture finale. En concevant cet environnement, nous serons en mesure d'effectuer des expériences d'injection de fautes pour évaluer les effets des SBU et MBU provoqués par des fautes transitoires dans les registres du processeur et dans le chemin de données, et par conséquent d'analyser la robustesse contre l'inversion d'un bit unique.

En pratique, le modèle VHDL au niveau RTL utilisé lors de la synthèse du circuit n'est pas utilisé pour la simulation d'injection de fautes. Afin de permettre une simulation très rapide, et donc permettre à un grand nombre de campagnes de simulation d'être réalisées dans un délai minimal, des outils dédiés ont été développés en C++ pour remplacer le modèle original de simulation « événementielle » sur lequel le VHDL est basé par un modèle de simulation « cyclique » plus rapide qui, de plus, est très bien adapté aux conceptions synchrones [CHL97]. Pour la simulation, ce modèle « cyclique » strictement équivalent en C++ a remplacé les modèles originaux développés en VHDL au niveau RTL.

Le point de départ de la conception de l'environnement est de définir comment décrire une manière de reproduire les fautes transitoires : quand les reproduire, où les affecter et que faut-il changer ? Nous avons choisi une approche non-déterministe de déclenchement de faute lors d'une séquence fixe où les basculements de bits peuvent être provoqués aléatoirement dans le cœur et le journal.

Les étapes de base d'une campagne d'injection de fautes sont présentées dans la figure 6.4. Le simulateur développé en C++ injecte les fautes dans le modèle du processeur en choisissant aléatoirement le ou les bits parmi l'ensemble des bits qui forment les registres. Après l'injection de faute, la simulation est arrêtée après 2 cycles et les circuits d'autocontrôle indiquent si l'erreur a été détectée ou pas. Si elle a été détectée, le compteur s'incrémente puis une nouvelle campagne de simulation commence avec un nouveau profil d'injection de fautes, comme illustré par la figure 6.4. Enfin, un rapport du nombre total d'erreurs injectées sur les erreurs détectées est généré.

Deux types de méthodes d'injection ont été réalisés : une campagne visant à injecter des erreurs bits uniques (SBU), des motifs d'erreurs aléatoires doubles et triples (MBU), et une autre campagne visant à injecter des erreurs aléatoires sévères (de 1 à 8 bits). Les résultats sont présentés dans les graphiques des figures 6.5, 6.6, 6.7 et 6.8, respectivement.

Pour le scénario 1, la figure 6.5 montre que le processeur peut détecter 100 % des erreurs bits

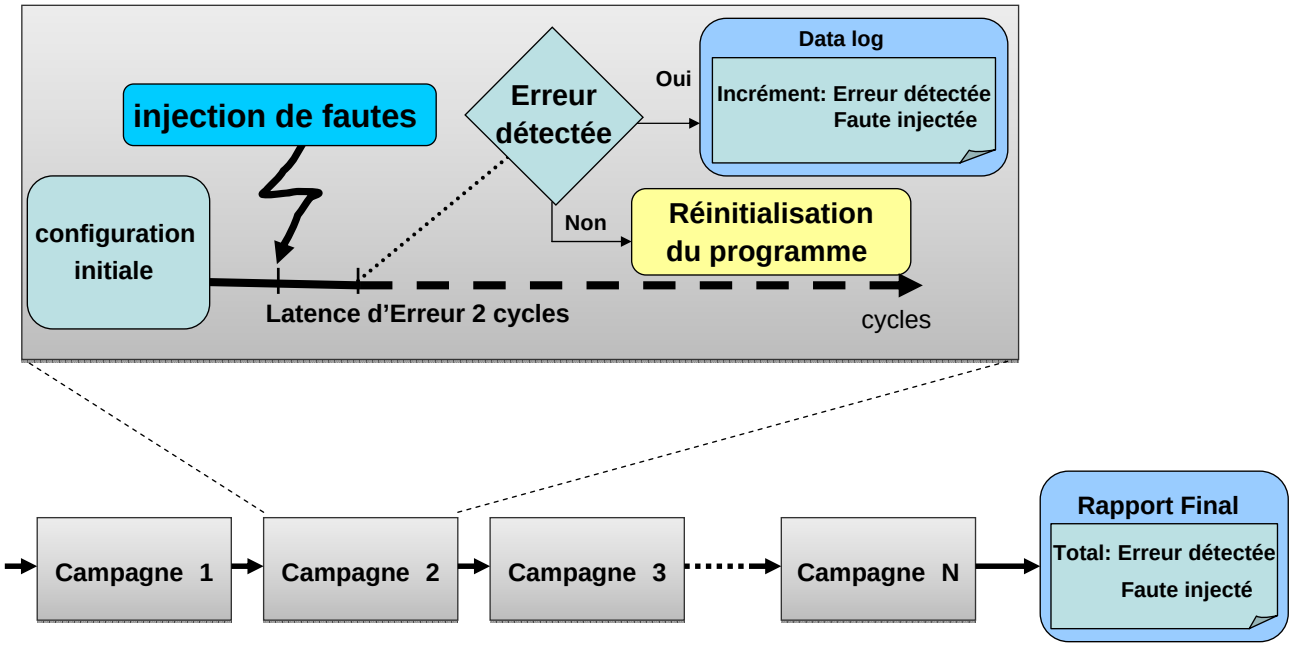


FIGURE 6.4 – Dispositif expérimental

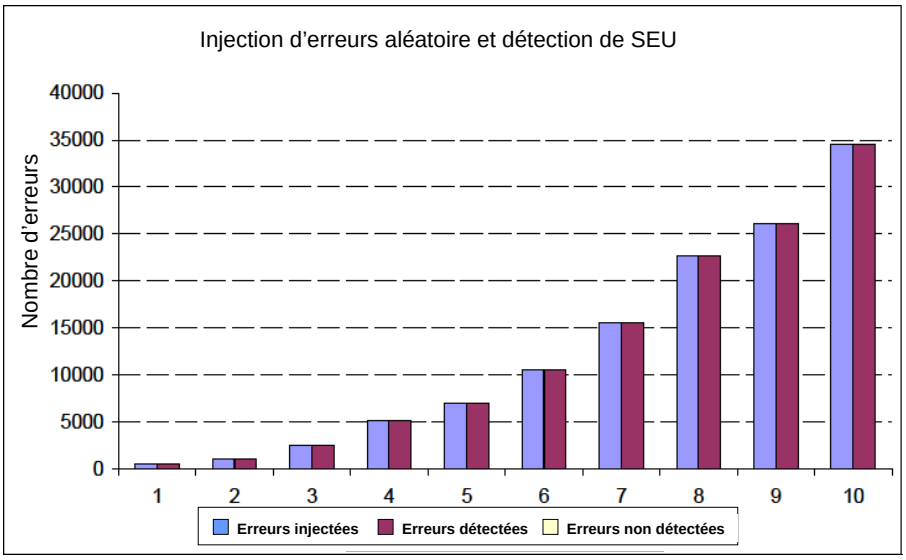


FIGURE 6.5 – Injection d'erreur simple (SBU)

uniques injectées. La détection des erreurs doubles et triples s'est effectuée avec des taux supérieurs à 60 % et 78 % respectivement, comme indiqué dans les figures 6.6 et 6.7. Dans le scénario 2 où les modèles sévères sont utilisés (1 à 8 bits, aléatoirement), le taux de détection reste encore significatif avec une valeur supérieure à 36 % pour toutes les configurations, comme le montre la figure 6.8.

Il est intéressant de noter que, en utilisant des codes de détection très simples dans le cœur, la couverture d'erreurs est toujours de 100 % pour les SBU. En tenant compte de la faible quantité de registres à protéger dans le cœur du processeur et du fait que sa surface représente seulement une fraction de la surface totale du processeur tolérant aux fautes, l'utilisation de codes plus évolués dans

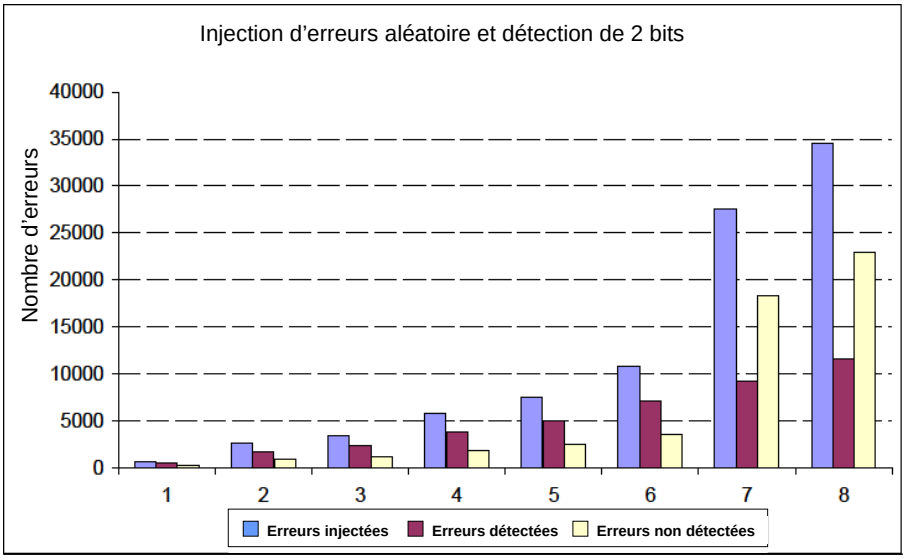


FIGURE 6.6 – Injection d'erreurs doubles

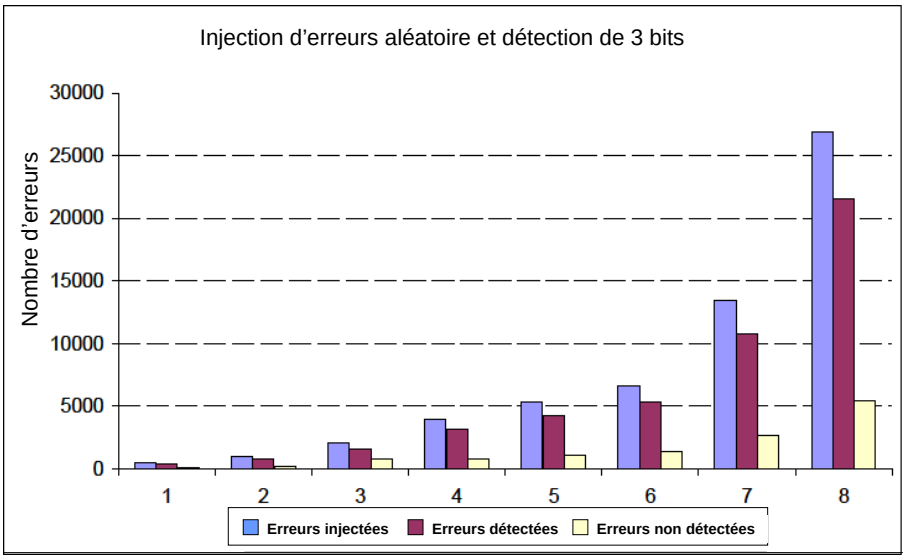


FIGURE 6.7 – Injection d'erreurs triples

le cœur pourra probablement améliorer le niveau de tolérance aux fautes, sans avoir un impact très important sur la surface.

Cela tend à prouver que l'approche proposée de conception du processeur tolérant aux fautes est une approche pertinente. Il est cependant nécessaire d'évaluer l'impact de l'augmentation des taux d'erreur sur les performances.

6.4 Dégradation des performances due à la réexécution

Pour mesurer l'impact des fautes transitoires sur les performances du système, nous avons évalué la dégradation des performances sur différents ensembles de benchmarks grâce à des simulations. Le nombre moyen de cycles d'horloge par opération (*clock per operation*, CPO) a été mesuré pour

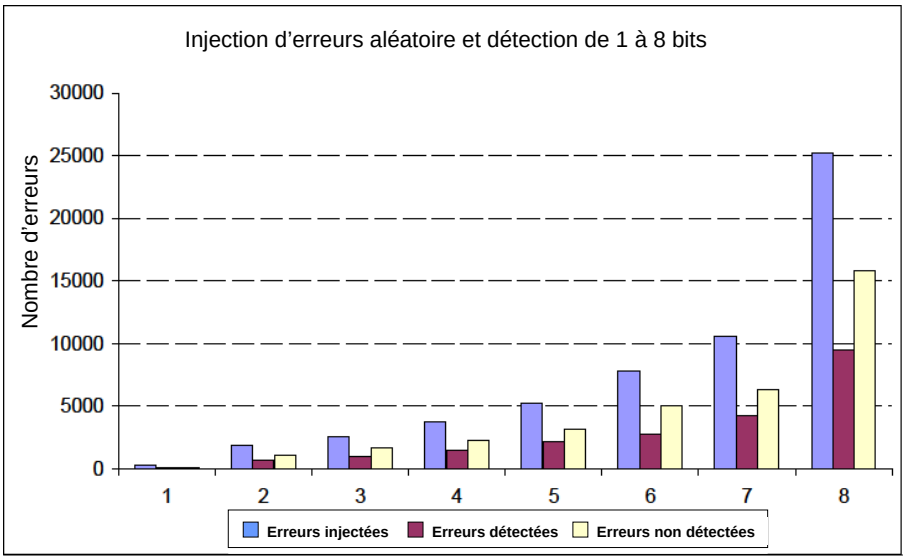


FIGURE 6.8 – Injection d'erreurs sévères (1 à 8 bits, aléatoirement).

différentes valeurs de EIR, en tant qu'indicateur de performance (plus la valeur est élevée, plus les performances sont faibles) et donc de dégradation des performances pour différentes conditions d'injection d'erreur.

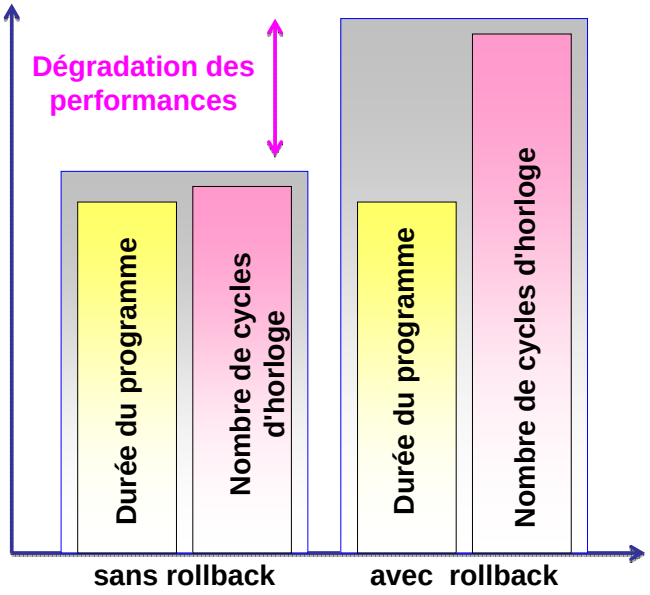


FIGURE 6.9 – Dégradation des performances due à la réexécution

Dans le processeur à pile pipeliné et journalisé, toutes les instructions sont exécutées en un cycle d'horloge à l'exception de `STORE`. Par conséquent, le nombre moyen de cycles d'horloge par opération (CPO) ou le nombre de cycles d'horloge par instruction exécutée est de un. Toutefois, dans le cas d'une détection d'erreur, le *rollback* est exécuté, ce qui augmente la pénalité temporelle globale. Plus le taux de *rollback* est grand, plus la valeur moyenne de CPO est élevée, car il faudra plus de cycles d'horloge pour compléter la tâche requise (cf. figure 6.9). En d'autres termes, plus la valeur moyenne

de CPO est élevée, plus la performance globale est faible.

Les benchmarks ont déjà été discutés dans la section 3.6. Ici, le tableau 6.1 résume les profils de pourcentage de lecture et d'écriture en mémoire principale pour chaque groupe d'instructions exécutées par le cœur. Notez que le jeu d'instructions du SCPC contient 36 instructions parmi lesquelles 23 impliquent la lecture ou l'écriture en mémoire.

TABLE 6.1 – Profils de lecture/écriture des groupes de benchmarks

Groupe	Lecture	Écriture
I	45 %	39 %
II	57 %	38 %
III	50 %	38 %

6.4.1 Évaluation de la dégradation de performance

L'objectif est de mesurer l'effet de la réexécution sur la longueur de la séquence. Nous avons tracé les graphes du nombre moyen de cycles par opération CPO par rapport au taux d'injection d'erreur EIR pour les différents benchmarks. Les figures 6.10, 6.11 et 6.12 présentent les résultats des benchmarks des groupes I, II, et III respectivement, pour différentes longueurs de séquences (SD=10, 20, 50 et 100).

Dans ces graphes, le nombre de cycles d'horloge par opération a été tracé en fonction du taux EIR. Ici, la pénalité de chargement des éléments sensibles SE n'a pas été considérée. Les erreurs ont été injectées dans le processeur et le journal à différents taux. L'analyse des graphiques montre que les courbes tendent à se confondre pour les valeurs les plus faibles d'EIR. Ceci est logique puisque, en l'absence d'erreur, aucune pénalité de temps supplémentaire due au *rollback* n'est induite, quel que soit le benchmark utilisé.

Dans la figure 6.10, le déplacement du point A au point B correspond à une augmentation de 10 % du taux d'erreur. L'augmentation correspondante en CPO reste faible (presque inchangée pour SD=10 et 20, seulement 1,1 pour SD=50 et 1,6 pour SD=100), ce qui signifie que la dégradation de performances est faible voire nulle. De même, le passage de A à C correspond à une augmentation des taux d'erreur de 100 % : le CPO reste très faible pour SD=10, et inférieur à 2 pour SD=20 et 50. Des observations similaires peuvent être faites à partir des graphiques des figures 6.11 et 6.12. Avec une augmentation du taux d'erreur de 100 %, la pénalité temporelle pour des séquences courtes reste raisonnable ce qui signifie une bonne performance.

Avec un EIR plus important, les séquences les plus courtes sont celles qui présentent la plus faible pénalité. Cela est également cohérent avec les résultats prédits. En fait, pour un taux d'erreur donné, le risque que la séquence soit invalidée est plus élevé pour une séquence longue, conduisant à un taux de *rollback* plus élevé.

En prenant en compte le fait que l'architecture choisie pour le cœur nécessite peu de temps pour la

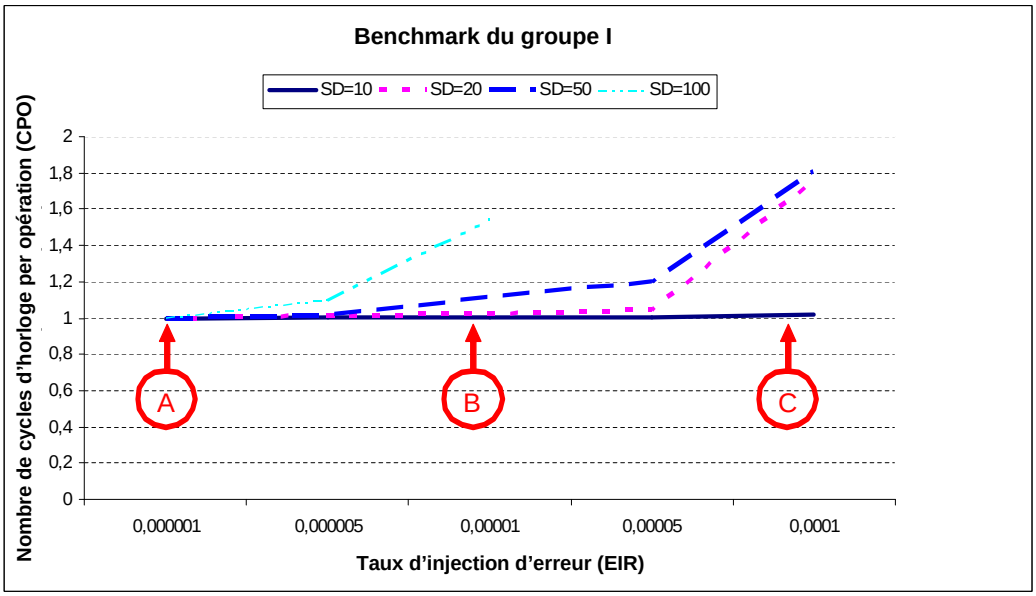


FIGURE 6.10 – Courbes de simulation pour le groupe I.

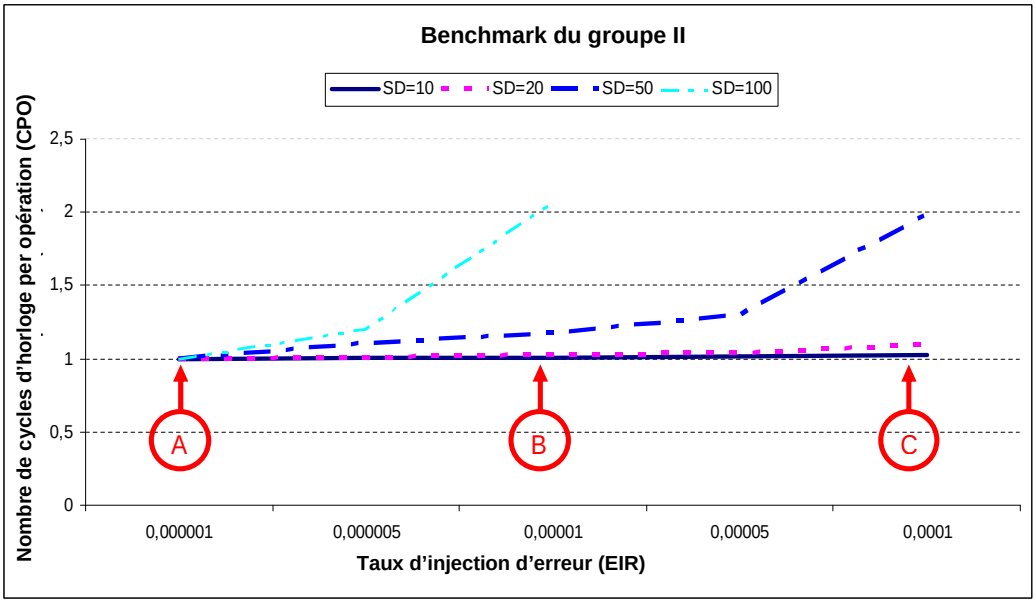


FIGURE 6.11 – Courbes de simulation pour le groupe II

sauvegarde des éléments sensibles, il est possible de sélectionner une séquence courte pour conserver un bon niveau de performance. De plus, cela permet de choisir une plus faible profondeur de journal et donc une consommation de surface réduite. Cela permet également de réduire davantage le risque que des erreurs se cumulent dans le journal et conduisent à une erreur non récupérable.

6.5 Effet de l'injection d'erreur sur les taux de *rollback*

Une augmentation du taux de *rollback* est un facteur limitant la performance en raison de la pénalité due à la réexécution des séquences. En conséquence, dans cette section, nous analysons

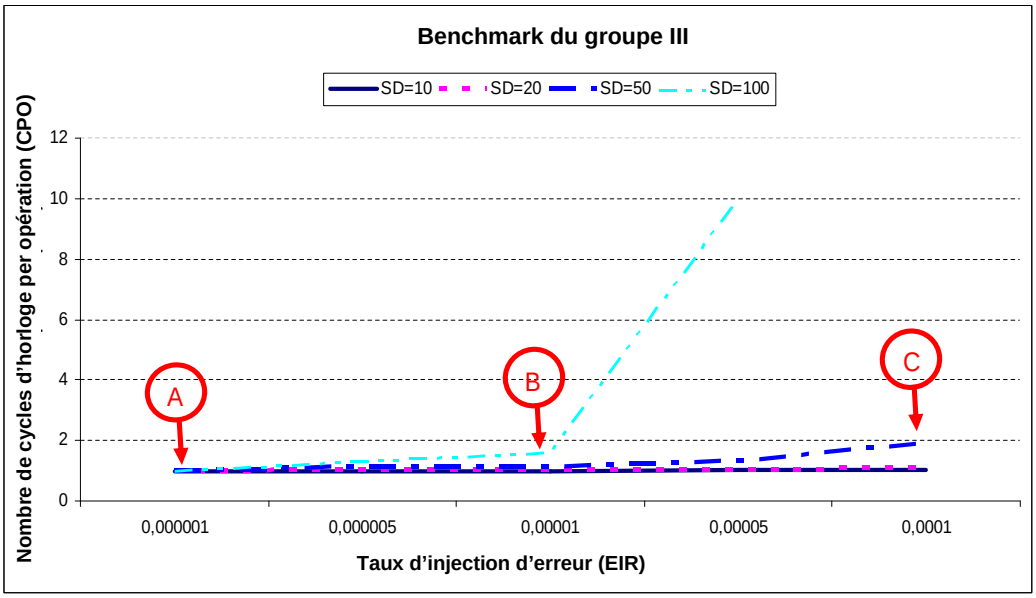


FIGURE 6.12 – Courbes de simulation pour le groupe III.

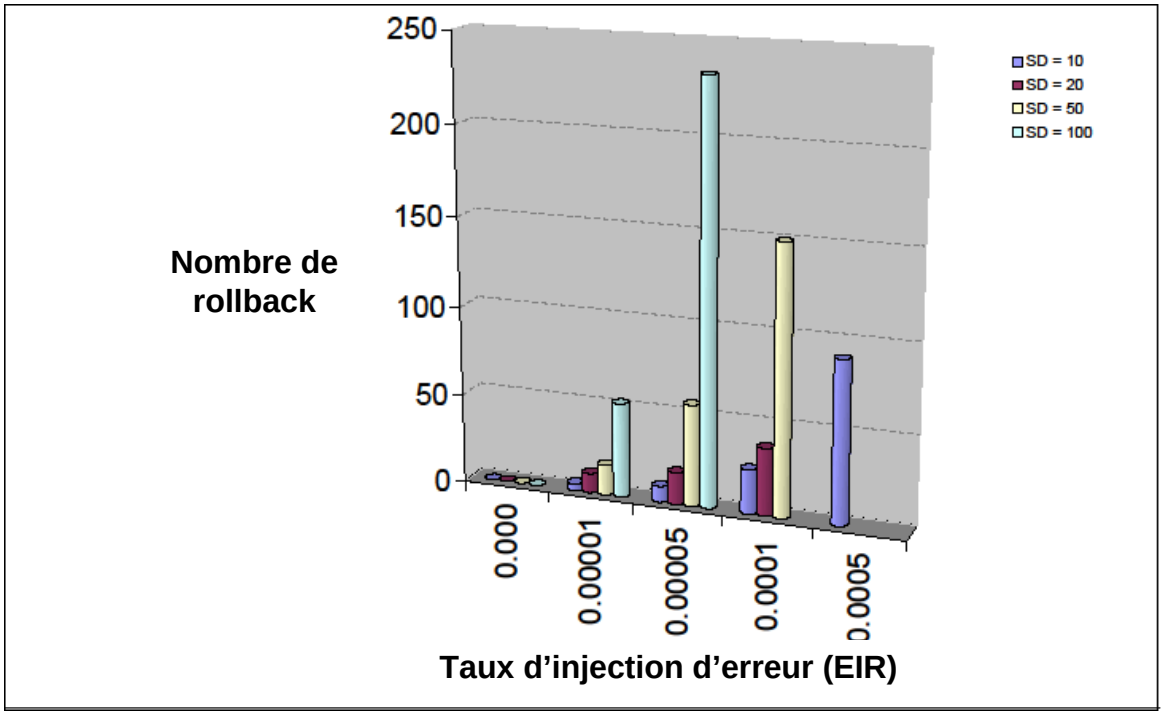


FIGURE 6.13 – Effet de l'EIR sur le rollback pour benchmarks du groupe I.

l'augmentation du taux de *rollback* due à l'augmentation du taux d'erreurs injectées. En fait, pour des EIR élevés, le taux de réexécution va également augmenter ce qui diminue d'autant la performance globale. Toutefois, si la probabilité d'erreur est connue, alors il est possible de trouver le nombre optimal de points de contrôle et de *rollbacks* possibles [VSL09]. Dans les systèmes réels, la probabilité d'erreur n'est pas connue à l'avance et reste difficile à estimer.

Dans le mécanisme de *rollback*, il y a deux facteurs qui limitent la performance : (i) le temps nécessaire pour sauvegarder ou recharger les éléments sensibles, et (ii) la longueur de la séquence

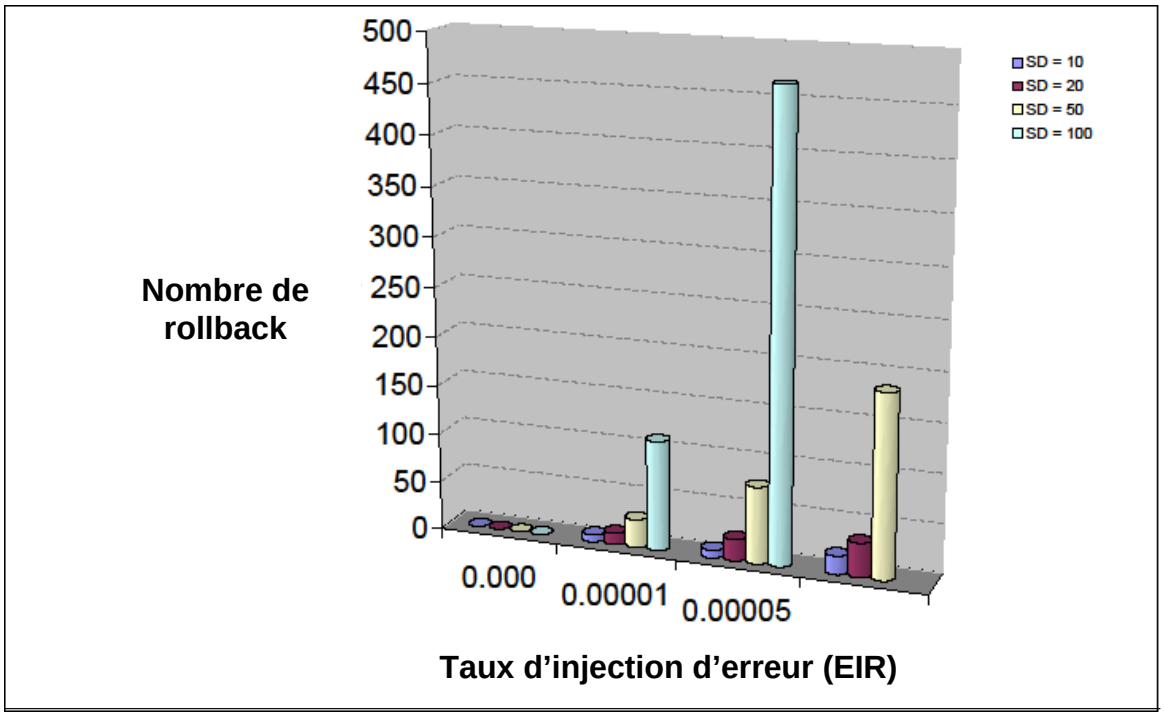


FIGURE 6.14 – Effet de l'EIR sur le rollback pour benchmarks du groupe II.

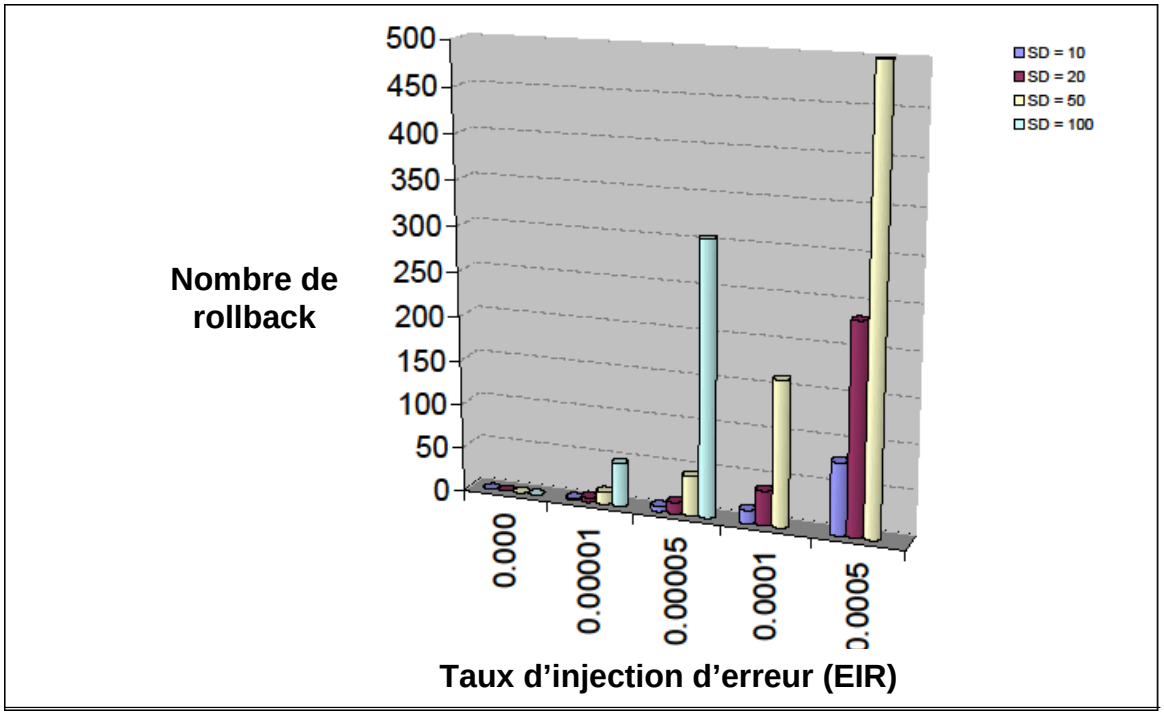


FIGURE 6.15 – Effet de l'EIR sur le rollback pour benchmarks du groupe III.

(SD). Si nous avons besoin de réduire la pénalité due au rechargement des SE, il est préférable d'avoir de longues séquences afin que le nombre global de chargements et de sauvegardes des éléments sensibles soit plus faible. Ce comportement doit être confirmé par l'injection d'erreur artificielle.

Par conséquent, nous avons injecté artificiellement des erreurs dans le processeur tolérant aux fautes pour observer son effet sur le mécanisme de *rollback* comme indiqué dans les figures 6.13,

6.14 et 6.15. Note : pour des durées de séquences élevées comme 50 ou 100, le nombre de *rollbacks* pour des taux d'injection élevés est absent parce que les valeurs obtenues sortent de l'échelle de l'axe des Y du graphique. À partir des courbes de simulation, il a été démontré que pour les taux d'erreurs faibles le taux de *rollback* est également faible et inversement. Par ailleurs, pour des taux d'erreurs plus élevés, l'effet du *rollback* est dominant dans les longues séquences. Ainsi, il y aura un plus grand nombre de *rollbacks* qui, encore une fois, donneront lieu à des pénalités temporelles et limiteront la performance globale.

Par conséquent, il est conseillé d'utiliser des séquences plus longues pour des faibles taux d'erreurs, et des séquences courtes pour des taux d'erreur plus élevés. Puisqu'on ne peut proposer une durée optimale de séquence uniquement si l'application finale est connue, la longueur de la séquence sera un paramètre défini par l'utilisateur qui pourra être ajusté selon l'environnement externe.

6.6 Comparaison avec le LEON 3 FT

Le processeur tolérant aux fautes LEON 3 FT a été présenté précédemment dans la section 2.4. Dans cette partie, nous comparons le schéma de protection du LEON 3 FT avec le processeur à pile journalisé. Ceci sera une comparaison qualitative.

Le LEON 3 FT met l'accent sur la protection du stockage de données et non pas sur la fonctionnalité de l'architecture. Les techniques mises en avant sont l'utilisation de codes ECC et la duplication des états internes. La plupart des registres permettent une détection d'erreurs de 2 bits alors que peu d'entre eux permettent une détection d'erreurs de 4 bits. Il n'y a pas de protection du chemin de données, de la fonctionnalité de l'ALU ni de l'unité de contrôle.

D'autre part, dans le processeur à pile journalisé tolérant aux fautes, l'objectif est d'avoir une protection globale de l'architecture. Dans le processeur, une détection d'erreur d'un seul bit est mise en œuvre, et le journal peut détecter une erreur de deux bits. Toutefois, dans les versions futures, une plus grande considération sera portée sur des codes à couverture plus élevée pour protéger le cœur du processeur. Le chemin de données est protégé, et l'ALU et le chemin de contrôle peuvent être protégés sans surcoût matériel. En résumé, le processeur journalisé tolérant aux fautes est en phase de développement mais montre déjà des caractéristiques intéressantes, bien qu'il nécessite des optimisations supplémentaires du point de vue de la protection.

6.7 Conclusion

Dans ce chapitre, nous avons validé la conception d'un processeur à pile journalisé tolérant aux fautes. Lors de la validation, nous avons évalué différents paramètres tels que la capacité d'autovérification, l'impact sur les performances temporelles et l'augmentation du taux de *rollback* dus à l'injection d'erreurs. Enfin, le modèle proposé a été comparé avec le processeur LEON 3 FT.

Pour des injections d'erreurs simples, 100 % des erreurs ont été détectées dans plusieurs confi-

gurations expérimentales. De même, avec des injections d'erreurs doubles ou triples, le pourcentage moyen de détection était d'environ 60 % et 78 % respectivement. Selon les résultats obtenus avec les modèles d'erreurs beaucoup plus graves (jusqu'à 8 bits erronés), la correction est toujours possible avec un taux de correction assez élevé d'environ 36 %.

Les résultats des tests de dégradation de performance ont également montré des résultats satisfaisants. L'architecture proposée offre une performance plutôt bonne, même en présence de taux d'erreurs élevés. Avec des taux d'erreurs importants, la pénalité temporelle peut rester raisonnable en utilisant des longueurs de séquences faibles. En pratique, il est conseillé d'utiliser des longues séquences pour les taux d'erreurs faibles, et des séquences courtes pour des taux d'erreurs plus élevés. Connaissant l'application finale et le profil d'erreur lié à l'environnement d'exécution, il est possible de choisir la durée la plus appropriée pour les séquences, ce qui est laissé comme un paramètre générique dans les modèles synthétisés.

CONCLUSION GÉNÉRALE

Conclusion générale

Avec les évolutions technologiques prévues, les erreurs *softs* dans les circuits électroniques deviennent un problème majeur pour la conception de systèmes numériques complexes, particulièrement dans les applications avec une exigence de sécurité critique. En effet, les progrès actuels en nanotechnologie, largement basés sur la diminution des dimensions des composants, la réduction de la tension d'alimentation et l'augmentation de la fréquence d'horloge, abaissent les marges de bruit. En conséquence, la sensibilité des circuits numériques aux particules de haute énergie et aux perturbations électromagnétiques augmente très rapidement, rendant la probabilité d'occurrence de SEU (SBU et MBU) très élevée, non seulement dans l'espace mais aussi pour les applications terrestres. Ainsi, le fait de prendre en compte, dès le début de toute conception électronique, le risque croissant que ces fautes transitoires surviennent devient très vite un besoin critique.

Pour assurer son bon fonctionnement, même en présence de fautes transitoires, il est nécessaire qu'un système comporte une certaine capacité de tolérance aux fautes. Parallèlement à la question de la tolérance aux fautes, la demande pour des systèmes plus grands, plus rapides, plus complexes et plus flexibles et, aussi et surtout, plus faciles à concevoir, est perpétuelle. En collaboration avec les moyens améliorés de communication intégrée sur puce (*Network on Chip*, NOC), les possibilités accrues d'intégration dans les circuits électroniques modernes permettent désormais de regrouper toutes les fonctionnalités d'un système complet dans une seule puce (*System on Chip*, SoC). Parmi les développements récents, le paradigme de conception MPSoC est devenu très populaire pour sa capacité à fournir à la fois une puissance de calcul importante et de la flexibilité. Il rassemble un grand nombre de processeurs, ou nœuds de calcul, reliés par un NoC, moyen de communication inter-nœuds. Un MPSoC n'étant pas naturellement à l'abri de fautes transitoires, un objectif évident est de lui apporter une capacité de tolérance aux fautes, et donc un processeur tolérant aux fautes pouvant être utilisé comme nœud de traitement.

Le travail qui a été présenté dans cette thèse est dédié à la conception d'un tel processeur tolérant aux fautes utilisant une nouvelle approche architecturale, les objectifs de conception abordés étant notamment un niveau élevé de protection contre les fautes transitoires ainsi que des performances et un surcoût en surface raisonnables. Il a été clair dès le début que des contraintes sévères concernant la consommation de la surface devraient s'appliquer à la conception architecturale du nœud de traitement afin de correspondre à l'objectif de parallélisation massive, tout en préservant autant que possible la performance du nœud.

Les concepts choisis pour être à la base de notre méthodologie de conception sont la capacité de

détection d'erreur concurrente en ligne, et le recouvrement d'erreur par réexécution, ou *rollback*. Au centre de la nouvelle architecture a trouve un cœur de processeur autocontrôlé et un mécanisme de journalisation matériel. Le cœur du processeur, appartenant à la classe MISC au lieu des classiques RISC ou CISC, est un processeur autocontrôlé inspiré du processeur à pile canonique [KJ89], capable d'offrir un assez bon niveau de performance avec une quantité de matériel nécessaire limitée. La simplicité architecturale – faible quantité de ressources logiques et de stockage interne – et la grande compacité du code sont des caractéristiques importantes favorables à la capacité d'autocontrôle et à l'implantation du recouvrement par *rollback*. À côté du cœur du processeur, un journal matériel autocontrôlé dédié au mécanisme de journalisation empêche la propagation des erreurs du cœur du processeur vers la mémoire principale et limite l'impact du *rollback* sur la performance temporelle. Parmi les hypothèses sous-jacentes, la mémoire principale est supposée être sûre, c'est-à-dire que les données sont supposées y être conservées de manière fiable sans aucun risque de corruption.

À l'occurrence d'une faute transitoire, les données peuvent être corrompues dans le processeur. Ces erreurs peuvent être détectées dans le cœur de processeur, mais non corrigées. Ainsi, des données erronées pourraient se propager hors du cœur du processeur et atteindre la mémoire sûre sans l'intervention du journal matériel. La mémoire principale serait dans ce cas un lieu non sûr et l'implantation d'un mécanisme logiciel de recouvrement serait assez pénible, avec la nécessité d'une redondance de données importante dans la mémoire. Les techniques classiques fonctionnant avec le *rollback* opèrent par *check point* : à des intervalles de temps réguliers, l'état du processeur et les données produites sont sauvegardés, permettant un *rollback* jusqu'au point de sauvegarde en cas de détection d'erreur. La durée de séquence la mieux adaptée, c'est-à-dire la distance entre deux points de vérification, dépend des contraintes de l'application et des taux d'occurrence d'erreur. Bien qu'une durée de séquence plus longue puisse limiter l'impact du mécanisme de *rollback* sur les performances temporelles en l'absence d'erreurs, elle nécessite un journal matériel plus grand et, de plus, augmente le risque d'activation du *rollback* en cas d'occurrence d'erreur.

Les données produites dans la séquence en cours peuvent être mises à l'écart en cas de détection d'erreur car elles peuvent être régénérées à partir du dernier point d'enregistrement. À la validation de séquence, sans occurrence d'erreur dans la séquence se terminant, les données concernées sont validées et doivent être transférées à la mémoire sûre. Les techniques de codage de contrôle d'erreur sont utilisées pour détecter les erreurs dans le cœur du processeur et dans la partie des données non validées du journal, et utilisées pour corriger les erreurs dans la partie des données validées du journal.

L'architecture de processeur tolérant aux fautes a été modélisée en VHDL au niveau RTL, puis synthétisée en utilisant Quartus II d'Altera pour déterminer les exigences en surface et la fréquence de fonctionnement maximale. Des campagnes d'injections d'erreurs simulées ont été réalisées pour déterminer l'efficacité de la stratégie proposée de tolérance aux fautes selon différents scénarios de fautes (variation du taux d'erreur et des profils d'erreur) et des durées de séquence variables.

La capacité d'autocontrôle du processeur tolérant aux fautes a été évaluée pour les SBU, motifs d'erreurs sur 1 bit, et les MBU, motifs d'erreurs de 2 à 8 bits dans un seul mot de données de 16 bits. Considérant les SBU, 100 % des erreurs sont détectées et la récupération d'erreur est 100 %

pour des taux élevés d'injection d'erreurs. Avec les modèles à 2 et 3 bits, le pourcentage moyen de détection était d'environ 60 % et 78 %, respectivement. Lorsque des conditions difficiles sont considérées avec des modèles d'erreurs d'au maximum 8 bits, la correction est encore possible avec des taux de correction d'environ 36 %.

De même, la dégradation des performances due à l'injection d'erreur a été évaluée. Le recouvrement d'erreur étant basé sur l'exécution du *rollback* lors de la détection d'erreur, les instructions de la séquence défectueuse sont ré-exécutées à partir des états précédents sauvegardés, mais en ajoutant une pénalité en temps, c'est-à-dire une dégradation des performances. Des taux plus élevés d'injection d'erreur induisent des taux plus élevés de *rollback*, entraînant ainsi une baisse des performances. L'analyse de la courbe de dégradation des performances mesurée montre que l'architecture proposée offre de bonnes performances, même en présence de taux d'erreurs élevés. Elle montre également que la durée de séquence optimale dépend du taux moyen d'injection d'erreurs qui devrait être ajusté en fonction de l'environnement extérieur de l'application.

En pratique, les résultats expérimentaux démontrent que le principe de journalisation peut être assez efficace sur une architecture de cœur de processeur à piles, et mérite plus d'effort de recherche pour améliorer les performances et la capacité de protection.

Les futurs travaux peuvent être divisés en deux aspects : la protection et la performance. Du point de vue de la protection, il est nécessaire d'améliorer la couverture d'erreurs dans le cœur du processeur. Actuellement, la parité simple peut uniquement détecter les erreurs affectant un nombre impair de bits. Le défi consiste à chercher des codes peu coûteux d'un point de vue matériel. Par ailleurs, l'opcode – dans les circuits de contrôle – peut être protégé par ECC. Dans la méthodologie reposant sur une architecture MISC à pile, il y a 37 instructions, avec un opcode de 8 bits. Il y a donc la possibilité d'utiliser des bits de redondance sans surcoût.

Du point de vue des performances, l'optimisation architecturale est nécessaire, principalement pour le journal matériel. Le processeur proposé présente un chemin critique dans le circuit de correction d'erreurs et écrit en mémoire principale. Si cette tâche est divisée en deux étages de pipeline, alors la performance globale peut être nettement améliorée. Un autre aspect possible est de combattre la chute de performances due aux branchements conditionnels. La méthode mise en œuvre consiste à charger les deux destinations possibles du saut dans le buffer d'instruction.

Sur le long terme, la poursuite de ce travail doit être consacrée à l'intégration de cette architecture de processeur tolérant aux fautes en tant que bloc d'un MPSoC tolérant aux fautes.

Annexe A

Processeur à pile canonique

Le processeur à pile canonique [KJ89] a été choisi pour développer le cœur de processeur tolérant aux fautes. Ses caractéristiques ressemblent essentiellement à celles de la seconde génération de machines à pile, plus économiques que celles de la première génération. Dans cette section, nous présentons brièvement la construction d'une machine à pile canonique, présentation utile pour comprendre les similitudes et les différences avec le processeur proposé.

La figure A.1 montre le schéma du processeur à pile canonique. Chaque bloc représente une des ressources logiques qui incluent : le bus de données, la pile de donnée (DS), la pile de retour (RS), l'unité arithmétique et logique (ALU), le registre sommet de la pile (TOS), le compteur programme (PC), le registre d'adresse mémoire (MAR), le registre d'instruction (IR), et une unité d'entrée/sortie (I/O). Pour des raisons de simplicité la machine canonique a été représentée avec un seul bus de données, mais les processeurs réels peuvent en inclure plus d'un pour effectuer en parallèle la lecture des instructions d'une part, et leur exécution d'autre part (cf. figure A.1). De même, les processeurs réels peuvent posséder plus d'un chemin de données pour permettre de paralléliser le calcul avec la lecture des instructions.

La pile DS est un tampon qui fonctionne en mode LIFO (*Last In First Out*). Seules deux opérations – `PUSH` et `POP` – peuvent avoir lieu sur la pile. Lors d'un `PUSH`, les nouveaux éléments de données sont écrits sur la position la plus haute de la pile, et les anciennes valeurs sont décalées d'une position vers le bas. Lors d'un `POP`, la valeur présente sur le sommet de la pile est placée sur le bus de données et la cellule suivante de la pile est décalée d'un rang vers le haut et ainsi de suite. De même, la pile RS est également mise en œuvre selon le principe LIFO. La seule différence est que la pile de retour est utilisée pour stocker des adresses de retour des routines au lieu des opérandes des instructions.

Le bloc de mémoire programme possède à la fois un registre d'adresse mémoire et une quantité raisonnable de mémoire à accès aléatoire. Pour accéder à la mémoire, l'adresse de lecture ou d'écriture est d'abord écrite dans le MAR. Ensuite, lors du cycle suivant, soit la donnée présente sur le bus est écrite en mémoire à l'adresse pointée par le MAR (mode écriture), soit la donnée présente à l'adresse pointée par le MAR sera copiée sur le bus (mode lecture).

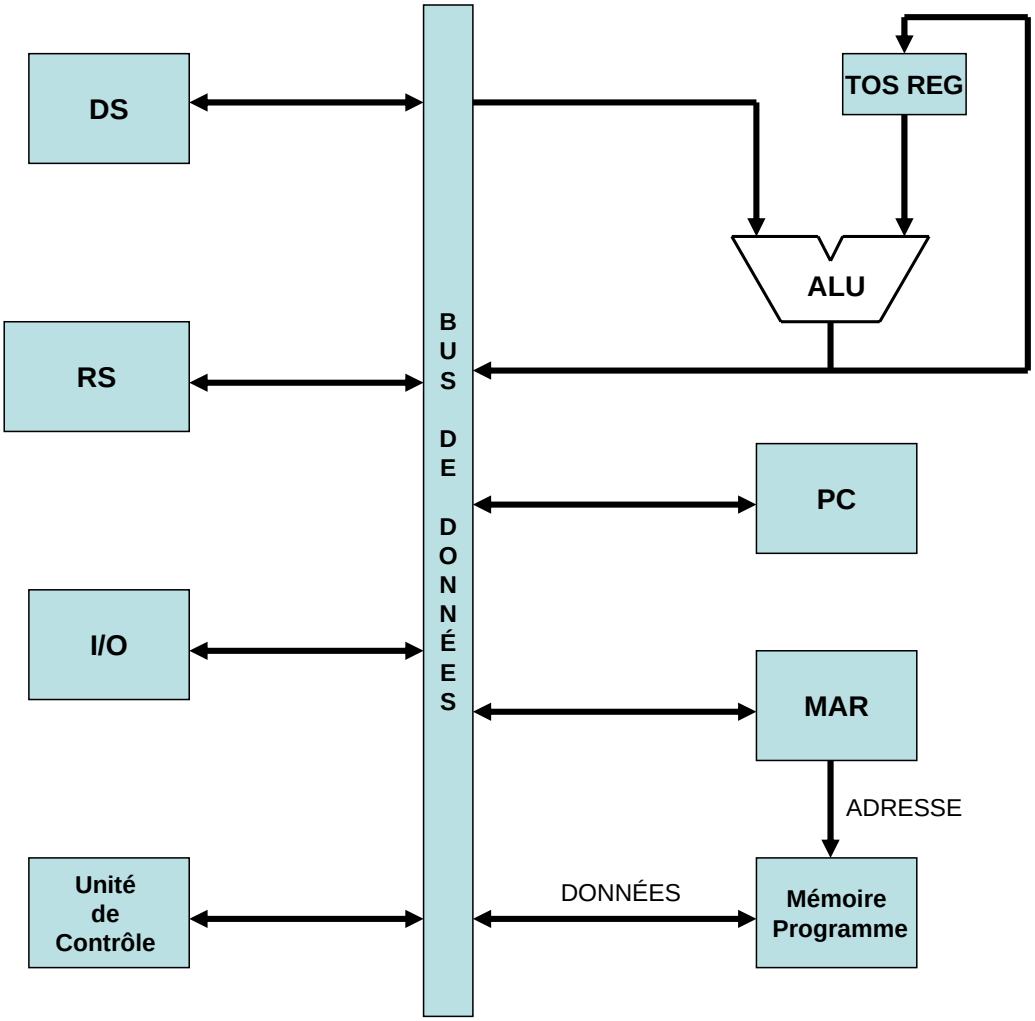


FIGURE A.1 – Machine à pile canonique [KJ89]

Annexe B

Jeu d'instructions du processeur à pile

Opérations arithmétiques et logiques

Les opérations arithmétiques et logiques sont montrées dans le tableau B.1. Elles sont les mêmes que dans la machine à pile canonique, mais elles ont été modifiées pour répondre à nos besoins. Certaines instructions supplémentaires comme l'addition avec retenue (*Addition with Carry*, `ADC`), la soustraction avec retenue (*Subtraction with Carry*, `SUBC`), le modulo (*Modulus*, `MOD`), la négation (*Negative*, `NEG`), la complémentation (`NOT`), l'incrémentation (*Increment*, `INC`), la décrémentation (*Decrement*, `DEC`) et le signe (`SIGN`) ont été ajoutées. Ces instructions supplémentaires offrent plus de flexibilité pour la programmation du processeur à Pile. Toutes les instructions sont décrites en pseudo code au niveau transfert de registre (RTL) et sont supposées être explicites.

Opérations de manipulation des piles

Les processeur à pile purs peuvent uniquement accéder aux deux sommets de la pile pour les opérations arithmétiques. Par conséquent, certaines instructions supplémentaires sont toujours nécessaires afin d'explorer les opérandes autres que les TOS, NOS ou TORS. Ici, de telles instructions comprennent la rotation (`ROT`), le transfert de RS vers DS (`R2D`), le transfert de DS à RS (`D2R`), et la copie de RS à DS (`CPR2D`). Les instructions `R2D`, `D2R` et `CPR2D` sont généralement utilisées pour intervertir les piles DS et RS. Le pseudo-code de toutes les instructions figurant dans ce tableau correspond à la version non pipelinée du modèle proposé.

Lecture et écriture en mémoire

Toutes les opérations arithmétiques et logiques sont effectuées sur les éléments de données de la pile. Ainsi, il doit y avoir un moyen de charger des informations sur la pile et de stocker des données dans la mémoire. Le pseudo code au niveau transfert de registre est dans le tableau B.3 ci-dessous.

TABLE B.1 – Opérations arithmétiques et logiques

Symbole	Instruction	Opérations
ADD	Addition	TOS $\Leftarrow$ TOS + NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
ADC	Addition avec retenue	TOS $\Leftarrow$ TOS + NOS NOS $\Leftarrow$ Cout
SUB	Subtraction	TOS $\Leftarrow$ TOS − NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
MUL	Multiplication	TOS $\Leftarrow$ TOS × NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
DIV	Division	TOS $\Leftarrow$ TOS ÷ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
MOD	Modulo	TOS $\Leftarrow$ TOS mod NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
AND	ET	TOS $\Leftarrow$ TOS & NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
OR	OU	TOS $\Leftarrow$ TOS | NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
XOR	OU exclusif	TOS $\Leftarrow$ TOS $\oplus$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP − 1
NEG	Négation	TOS $\Leftarrow$ −TOS
NOT	Complémentation	TOS $\Leftarrow$ not TOS
INC	Incrémentation	TOS $\Leftarrow$ TOS + 1
DEC	Décrémentation	TOS $\Leftarrow$ TOS − 1
SIGN	Signe	si (TOS$<$ 0) alors TOS $\Leftarrow$ 0× FFFF sinon si (TOS$>$ 0) alors TOS $\Leftarrow$ 0× 0000

Chargements littéraux

Il est nécessaire d'avoir un moyen de charger des constantes sur la pile. Les instructions pour ce faire comprennent LIT et DLIT qui peuvent charger respectivement un octet et un mot de données sur la pile DS, comme indiqué ci-dessous dans le tableau B.4.

TABLE B.2 – Opérations de manipulation des piles

Symbole	Instruction	Opérations
DROP	Drop	TOS $\Leftarrow$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
DUP	Duplication	DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS
SWAP	Échange	TOS $\Leftarrow$ NOS NOS $\Leftarrow$ TOS
OVER	Over	DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS TOS $\Leftarrow$ NOS NOS $\Leftarrow$ TOS
ROT	Rotation	TOS $\Leftarrow$ DS[DSP] NOS $\Leftarrow$ TOS DS[DSP] $\Leftarrow$ NOS
R2D	RS vers DS	TOS $\Leftarrow$ TORS NOS $\Leftarrow$ TOS DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS TORS $\Leftarrow$ RS[RSP] RSP $\Leftarrow$ RSP $-$ 1
CPR2D	Copie de RS vers DS	TOS $\Leftarrow$ TORS NOS $\Leftarrow$ TOS DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS
D2R	DS vers RS	TORS $\Leftarrow$ TOS DSP $\Leftarrow$ DSP $-$ 1 TOS $\Leftarrow$ NOS NOS $\Leftarrow$ DS[DSP] RSP $\Leftarrow$ RSP $+$ 1 RS[RSP] $\Leftarrow$ TORS
RET	Retour	IP $\Leftarrow$ TORS TORS $\Leftarrow$ RS [RSP] RSP $\Leftarrow$ RSP $-$ 1

Branchements conditionnels

Lors du traitement des données il est nécessaire de prendre des décisions. Le processeur doit avoir la possibilité d'effectuer des branchements conditionnels. Les sauts conditionnels peuvent dépendre de conditions différentes.

TABLE B.3 – Lecture et écriture en mémoire

Symbole	Instruction	Opérations
FETCH	Lecture	Mem_Addr $\Leftarrow$ TOS TOS $\Leftarrow$ Mem
STORE	Écriture	Mem_Addr $\Leftarrow$ TOS Mem $\Leftarrow$ NOS DSP $\Leftarrow$ DSP $-$ 1 TOS $\Leftarrow$ DS[DSP] DSP $\Leftarrow$ DSP $-$ 1 NOS $\Leftarrow$ DS[DSP]

TABLE B.4 – Chargements littéraux

Symbole	Instruction	Opérations
LIT d8	Écriture d'un octet dans TOS	DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ donnée (octet)
LIT d16	Écriture d'un mot dans TOS	DSP $\Leftarrow$ DSP $+$ 1 DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ donnée (mot)

TABLE B.5 – Branchements conditionnels

Symbole	Instruction	Opérations
ZBRA d	Saut vers d si TOS$= 0$	si (TOS$= 0$) alors IP $\Leftarrow$ IP $+$ d
SBRA d	Saut vers d si TOS< 0	si (TOS< 0) alors IP $\Leftarrow$ IP $+$ d

Appels de sous-programmes

Dans les processeurs à pile, la plupart des instructions sont exécutées entre TOS et NOS, mais dans ce type d'architecture, et afin d'améliorer la flexibilité des machines à base de pile, une pile RS est incluse, parallèlement à la pile DS. Le processeur proposé peut efficacement effectuer des appels des sous-programmes.

L'appel d'un sous-programme copie la valeur de PC dans TOS, et parfois il est également possible d'écrire directement une adresse connue dans TOS, comme indiqué ci-dessous dans le tableau B.6.

TABLE B.6 – Appels de sous-programmes

Symbole	Instruction	Opérations
`LBRA` a	Saut à l'adresse a	IP $\Leftarrow$ a
`CALL` a	Appel du sous-programme à l'adresse a	RSP $\Leftarrow$ RSP $+ 1$ RS[RSP] $\Leftarrow$ TORS TORS $\Leftarrow$ IP IP $\Leftarrow$ a

`PUSH` et `POP`

Les deux opérations de base de la pile sont `PUSH` et `POP`. Ces opérations sont disponibles pour la pile de données DS et la pile des retours RS. Elles sont détaillées dans le tableau B.7 ci-dessous.

TABLE B.7 – `PUSH` et `POP`

Symbole	Instruction	Opérations
`PUSH` DSP	Ajouter une donnée sur la pile DS	DSP $\Leftarrow$ DSP $+ 1$ DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ DSP
`POP` DSP	Retirer une donnée de la pile DS	DSP $\Leftarrow$ TOS TOS $\Leftarrow$ NOS DSP $\Leftarrow$ DSP $- 1$ NOS $\Leftarrow$ DS[DSP]
`PUSH` RSP	Ajouter une donnée sur la pile RS	DSP $\Leftarrow$ DSP $+ 1$ DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ RSP
`POP` RSP	Retirer une donnée de la pile RS	RSP $\Leftarrow$ TOS TOS $\Leftarrow$ NOS NOS $\Leftarrow$ DS[DSP] DSP $\Leftarrow$ DSP $- 1$

B.1 Opérations sur les données dans le processeur à pile

Un processeur à pile manipule les données en utilisant des opérations postfixes, également appelées « notation polonaise inverse ». Dans ce type d'opérations, les opérandes viennent avant l'opération qui agit sur les opérandes les plus récents.
Par exemple, si nous avons l'expression suivante : $(24 + 04) \times 82$
En représentation postfixe, cette expression devient : $82\ \ 24\ \ 04\ \ + \times$

Les expressions postfixes sont généralement plus compactes qu'en notation infixe. Un processeur à pile peut exécuter les expressions postfixes directement sans surcharger le compilateur.

Annexe C

Jeu d'instructions du processeur à pile pipeliné

Nous avons analysé les instructions multi-cycles pour analyser les conflits possibles entre les différentes instructions. Nous avons constaté que toutes les instructions multi-cycles peuvent être décomposées en deux étapes. La première étape consiste en DSP+1 ou RSP +1 selon le type d'instruction, alors que la seconde étape contient le reste de l'instruction. Ces instructions peuvent être reconnues par le code « 111 ». Grâce à un pipelining intelligent, une partie de la prochaine instruction (DSP+1/RSP+1) peut être pré-exécutée en même temps que l'instruction en cours. De cette façon, au cycle d'horloge suivant, la partie restante de l'instruction est exécutée. Ainsi, grâce au pipeline, le processeur peut exécuter toutes les instructions en un seul cycle d'horloge, sauf l'instruction `STORE` qui en nécessite deux après pipelining (avant le pipelining, l'instruction `STORE` nécessitait 3 cycles d'horloge). En fait, dans l'instruction `STORE` il est nécessaire d'exécuter deux fois DSP+1, ce qui ne peut être fait en un seul cycle. Le reste de l'instruction est exécuté au cycle d'horloge suivant.

Ainsi, toutes les instructions sont intelligemment décomposées en deux étapes afin que chaque instruction soit exécutée en un cycle d'horloge après l'implantation d'un pipeline à deux étages. La liste complète des instructions est donnée ci-dessous.

TABLE C.1 – Jeu d'instructions du processeur à pile pipeliné

Instructions	Premier étage	Deuxième étage
ADD	NOP	TOS $\Leftarrow$ TOS + NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
ADC	NOP	TOS $\Leftarrow$ TOS + NOS NOS $\Leftarrow$ Cout
SUB	NOP	TOS $\Leftarrow$ TOS $-$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
MUL	NOP	TOS $\Leftarrow$ TOS $\times$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
DIV	NOP	TOS $\Leftarrow$ TOS $\div$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
MOD	NOP	TOS $\Leftarrow$ TOS mod NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
AND	NOP	TOS $\Leftarrow$ TOS & NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
OR	NOP	TOS $\Leftarrow$ TOS | NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
XOR	NOP	TOS $\Leftarrow$ TOS $\oplus$ NOS NOS $\Leftarrow$ DS [DSP] DSP $\Leftarrow$ DSP $-$ 1
NEG	NOP	TOS $\Leftarrow -$TOS
NOT	NOP	TOS $\Leftarrow$ not TOS
INC	NOP	TOS $\Leftarrow$ TOS + 1
DEC	NOP	TOS $\Leftarrow$ TOS $-$ 1
SIGN	NOP	si (TOS$<$ 0) alors TOS $\Leftarrow$ 0$\times$ FFFF sinon si (TOS$>$ 0) alors TOS $\Leftarrow$ 0$\times$ 0000

TABLE C.2 – Opérations de manipulation des piles

Instructions	Premier étage	Deuxième étage
DROP	NOP	TOS ⇐ NOS NOS ⇐ DS [DSP] DSP ⇐ DSP−1
DUP	DSP ⇐ DSP+1	DS[DSP] ⇐ NOS NOS ⇐ TOS
SWAP	NOP	TOS ⇐ NOS NOS ⇐ TOS
OVER	DSP ⇐ DSP+1	DS[DSP] ⇐ NOS TOS ⇐ NOS NOS ⇐ TOS
ROT	NOP	TOS ⇐ DS[DSP] NOS ⇐ TOS DS[DSP] ⇐ NOS
R2D	DSP ⇐ DSP+1	TOS ⇐ TORS NOS ⇐ TOS DS[DSP] ⇐ NOS TORS ⇐ RS[RSP] RSP ⇐ RSP−1
CPR2D	DSP ⇐ DSP+1	TOS ⇐ TORS NOS ⇐ TOS DS[DSP] ⇐ NOS
D2R	RS[RSP] ⇐ TORS	TORS ⇐ TOS DSP ⇐ DSP−1 TOS ⇐ NOS NOS ⇐ DS[DSP] RSP ⇐ RSP+1
RET	NOP	IP ⇐ TORS TORS ⇐ RS [RSP] RSP ⇐ RSP−1

TABLE C.3 – Lecture et écriture en mémoire

Instructions	Premier étage	Deuxième étage
FETCH	Mem_Addr ⇐ TOS	TOS ⇐ Mem
STORE	Mem_Addr ⇐ TOS TOS ⇐ DS[DSP] DSP ⇐ DSP − 1	Mem ⇐ NOS NOS ⇐ DS[DSP] DSP ⇐ DSP − 1

TABLE C.4 – Chargements littéraux

Instructions	Premier étage	Deuxième étage
`LIT` d8	DSP $\Leftarrow$ DSP + 1	DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ donnée (octet)
`DLIT` d16	DSP $\Leftarrow$ DSP + 1	DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ donnée (mot)

TABLE C.5 – Branchements conditionnels

Instructions	Premier étage	Deuxième étage
`ZBRA` d	`NOP`	si (TOS$=0$) alors IP $\Leftarrow$ IP + d
`SBRA` d	`NOP`	si (TOS<0) alors IP $\Leftarrow$ IP + d

TABLE C.6 – Appels de sous-programmes

Instructions	Premier étage	Deuxième étage
`LBRA` a	`NOP`	IP $\Leftarrow$ a
`CALL` a	RSP $\Leftarrow$ RSP + 1	RS[RSP] $\Leftarrow$ TORS TORS $\Leftarrow$ IP IP $\Leftarrow$ a

TABLE C.7 – `PUSH` et `POP`

Instructions	Premier étage	Deuxième étage
`PUSH` DSP	DSP $\Leftarrow$ DSP + 1	DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ DSP
`POP` DSP	`NOP`	DSP $\Leftarrow$ TOS TOS $\Leftarrow$ NOS DSP $\Leftarrow$ DSP $-$ 1 NOS $\Leftarrow$ DS[DSP]
`PUSH` RSP	DSP $\Leftarrow$ DSP + 1	DS[DSP] $\Leftarrow$ NOS NOS $\Leftarrow$ TOS TOS $\Leftarrow$ RSP
`POP` RSP	`NOP`	RSP $\Leftarrow$ TOS TOS $\Leftarrow$ NOS NOS $\Leftarrow$ DS[DSP] DSP $\Leftarrow$ DSP $-$ 1

TABLE C.8 – Code et longueur des instructions

b_7 b_6 b_5	b_4 b_3 b_2 b_1 b_0	Type of instruction	Instruction Length
0 0 0	0 0 0 0 0	NOP	0-byte
1 0 0	0 0 0 0 0	ADD	1-byte
1 0 0	0 0 0 0 1	ADC	1-byte
1 0 0	0 0 0 1 0	SUB	1-byte
1 0 0	0 0 0 1 1	SUBC	1-byte
1 0 0	0 0 1 0 0	MUL	1-byte
1 0 0	0 0 1 0 1	DIV	1-byte
1 0 0	0 0 1 1 0	MOD	1-byte
1 0 0	0 0 1 1 1	AND	1-byte
1 0 0	0 1 0 0 0	OR	1-byte
1 0 0	0 1 0 0 1	XOR	1-byte
1 0 0	0 1 0 1 0	NEG	1-byte
1 0 0	0 1 0 1 1	NOT	1-byte
1 0 0	0 1 1 0 0	INC	1-byte
1 0 0	0 1 1 0 1	DEC	1-byte
1 0 0	0 1 1 1 0	SIGN	1-byte
1 0 0	0 1 1 1 1	DROP	1-byte
1 0 0	1 0 0 0 0	DUP	1-byte
1 0 0	1 0 0 0 1	SWAP	1-byte
1 0 0	1 0 0 1 0	OVER	1-byte
1 0 0	1 0 0 1 1	ROT	1-byte
1 0 0	1 0 1 0 0	R2D	1-byte
1 0 0	1 0 1 0 1	CPR2D	1-byte
1 0 0	1 0 1 1 0	D2R	1-byte
1 0 0	1 0 1 1 1	FETCH	1-byte
1 0 0	1 1 0 0 0	STORE	1-byte
1 0 0	1 1 0 0 1	PUSH_DSP	1-byte
1 0 0	1 1 0 1 0	POP_DSP	1-byte
1 0 0	1 1 0 1 1	PUSH_DSP	1-byte
1 0 0	1 1 1 0 0	POP_RSP	1-byte
1 0 1	0 0 0 0 0	LIT a	2-bytes
1 1 0	0 0 0 0 0	DLIT a	3-bytes
1 1 1	0 0 0 0 0	RET	1-byte + IP-change
0 0 1	0 0 0 0 1	ZBRA	2-bytes + IP-change
0 0 1	0 0 0 1 0	SBRA	2-bytes + IP-change
0 1 0	0 0 0 0 0	LBRA	3-bytes + IP-change
0 1 0	0 0 0 0 1	CALL a	3-bytes + IP-change

Annexe D

Liste des acronymes

ALU : *Arithmetic Logic Unit*, unité arithmétique et logique
ASIC : *Application Specific Integrated Circuits*, circuit intégré [à application] spécifique
BER : *Backward Error Recovery*, recouvrement d'erreur arrière
BIED : *Built-In Error Detection*, détection d'erreur intégrée
BPSG : *BoroPhosphoSilicate Glass*, verre de borophospho-silicates

CED : *Concurrent Error Detection*, détection d'erreur concurrente
CISC : *Complex Instruction Set Computer*, processeur à jeu d'instruction complexe
CMOS : *Complementary Metal Oxide Semiconductor*
CPO : *Clock Per Operation*, cycles d'horloge par opération
CPI : *Clock Per Instruction*, cycles d'horloge par instruction
CRC : *Cyclic Redundancy Codes*, codes à redondance cyclique

DCR : *Dual-Checker Rail*,
DED : *Double Error Detection*, détection d'erreur double
DM : *Dependable Memory*, mémoire sûre
DMR : *Dual Modular Redundancy*, redondance modulaire double
DS : *Data Stack*, pile de données
DSP : *Data Stack Pointer*, pointeur de pile de données
DWC : *Duplication With Comparison*, duplication avec comparaison
DWCR : *Duplication With Complement Redundancy*, duplication avec redondance complémentaire

ECC : *Error Control Coding*, codage de contrôle d'erreur
EDC : *Error Detecting Codes*, codes détecteurs d'erreur
EDCC : *Error Detecting and Correction Codes*, codes détecteurs et correcteurs d'erreur
EDP : *Error Detecting Processor*, processeur détecteur d'erreur
ESS : *Electronic Switching Systems*, système de commutation électroniques

FER : *Forward Error Recovery*, recouvrement d'erreur directe
FPGA : *Field Programmable Gate Array*, réseau de portes programmables par champ
FT : *Fault Tolerant*, tolérant aux fautes
FTMP : *Fault Tolerant Multi-Processor*, multiprocesseur tolérant aux fautes

HD : *Hamming Distance*, distance de Hamming
HDL : *High-level Description Language*, langage de description haut niveau
HW : *Hardware*, matériel

IEEE : *Institute of Electrical and Electronics Engineers*
IB : *Instruction Buffer*, buffer d'instructions
IBMU : *Instruction Buffer Management Unit*, unité de gestion du buffer d'instructions
IP : *Instruction Pointer*, pointeur d'instruction
ISA : *Instruction Set Architecture*, architecture du jeu d'instructions

LICM : *Laboratoire Interfaces Capteurs et Micro-électronique*,
LIFO : *Last In First Out*, Dernière entrée, première sortie (pile)

MBU : *Multiple Bit Upsets*, perturbations de plusieurs bits
MCU : *Multiple Cell Upsets*, perturbations de plusieurs cellules
MISC : *Minimum Instruction Set Computer*, processeur à jeu d'instructions minimal
MPSoC : *Multi-Processor System on Chip*, système sur puce multiprocesseur

NASA : *National Aeronautics and Space Agency*
NoC : *Network on Chip*, réseau sur puce
NOS : *Next Of data Stack*, sous-sommet de la pile de données

PC : *Program Counter*, compteur programme (ou compteur ordinal)

RAM : *Random Access Memory*, mémoire à accès aléatoire
REE : *Remote Exploration and Experimentation*, exploration et expérimentation à distance
RESO : *Redundant Execution with Shifted Operands*, exécution redondante avec opérandes décalés
RISC : *Reduce Instruction Set Computer*, processeur à jeu d'instructions réduit
RS : *Return Stack*, pile des retours
RSP : *Return Stack Pointer*, pointeur de pile des retours
RTL : *Register Transfer Level*, niveau transfert de registres

SCHJ : *Self-Checking Hardware Journal*, journal matériel autocontrôlé
SCPC : *Self-Checking Processor Core*, cœur de processeur autocontrôlé
SD : *Sequence Duration*, durée de la séquence
SE : *State determining Elements*, éléments déterminant de l'état [du processeur] (ou éléments sensibles)
SEB : *Single Event Burnout*, destruction isolée
SEE : *Single Event Effect*, effet isolé
SEGR : *Single Event Gate Rupture*, rupture de porte isolé
SEFI : *Single Event Functional Interrupt*, interruption fonctionnelle isolée
SEL : *Single Event Latchup*, verrouillage isolé
SEU : *Single Event Upset*, perturbation isolée
SET : *Single Event Transient*, effet transitoire isolé
SEC : *Single Error Correction*, correction d'erreur simple
SW : *Software*, logiciel
SIFT : *Software Implemented Fault Tolerance*, tolérance aux fautes implantée de façon logicielle
SoC : *System on Chip*, système sur puce
STAR : *Self-Testing and Repair*, autotest et réparation
TMR : *Triple Modular Redundancy*, redondance modulaire triple
TORS : *Top Of Return Stack*, sommet de la pile des retours
TOS : *Top Of data Stack*, sommet de la pile des données
UJ : *Un-validated Journal*, journal non validé
VP : *Validation Point*, point de validation
UVD : *Un-Validated Data*, donnée don validée
VD : *Validated Data*, donnée validée
VHDL : *VHSIC Hardware Description Language*, langage de description matérielle de circuits intégrés à très haute vitesse
VJ : *Validated Journal*, journal validé

Annexe E

Liste des publications

- Mohsin AMIN, Abbas RAMAZANI, Fabrice MONTEIRO, Camille DIOU, Abbas DANDACHE, "A Self-Checking HW Journal for a Fault Tolerant Processor Architecture," International Journal of Reconfigurable Computing 2011 (IJRC'11).

- Mohsin AMIN, Abbas RAMAZANI, Fabrice MONTEIRO, Camille DIOU, Abbas DANDACHE, "A Dependable Stack Processor Core for MPSoC Development," *XXIV Conference on Design of Circuits and Integrated Systems (DCIS'09)*, Zaragoza, Spain, November 18-20, 2009.

- Mohsin AMIN, Fabrice MONTEIRO, Camille DIOU, Abbas RAMAZANI, Abbas DANDACHE, "A HW/SW Mixed Mechanism to Improve the Dependability of a Stack Processor," *16th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'09)*, Hammamet, Tunisia, December 13-16, 2009.

- Mohsin AMIN, Camille DIOU, Fabrice MONTEIRO, Abbas RAMAZANI, Abbas DANDACHE, "Journalized Stack Processor for Reliable Embedded Systems," *1st International Conference on Aerospace Science and Engineering (ICASE'09)*, Islamabad, Pakistan, August 18-20, 2009.

- A. Ramazani, M. Amin, F. Monteiro, C. Diou, A. Dandache, "A Fault Tolerant Journalized Stack Processor Architecture," *15th IEEE International On-Line Testing Symposium (IOLTS'09)*, Sesimbra-Lisbonne, Portugal, 24–27 June 2009.

- Mohsin AMIN, Camille DIOU, Fabrice MONTEIRO, Abbas RAMAZANI, Abbas DANDACHE, "Error Detecting and Correcting Journal for Dependable Processor Core," *GDR System on Chip - System in Package (GDR-SoC-SiP'10)*, Cergy-Paris, France, 9-11 June 2010.

- Mohsin Amin, Camille Diou, Fabrice Monteiro, Abbas Ramazani, "Design Methodology of Reliable Stack Processor Core," *GDR System on Chip - System in Package 2009 (GDR-SoC-SiP'09)*, Orsay-Paris, France, 9-11 June 2010.

- Mohsin AMIN, “Self-Organization in Embedded Systems,” *2nd Winter School on Self Organization in Embedded Systems*, Schloss Dagstuhl, Germany, November 2007.

Table des figures

Liste des tableaux

Bibliographie

[ACC+93] J. Arlat, A. Costes, Y. Crouzet, J. C Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, page 913–923, 1993.

[Aer11] Aeroflex. Dual-Core LEON3FT SPARC v8 processor, 2011.

[AFK05] J. Aidemark, P. Folkesson, and J. Karlsson. A framework for node-level fault tolerance in distributed real-time systems. In *Proceedings of International Conference on Dependable Systems and Networks, 2005 (DSN'05)*, page 656–665, 2005.

[AHHW08] U. Amgalan, C. Hachmann, S. Hellebrand, and H. J Wunderlich. Signature Rollback-A technique for testing robust circuits. In *26th IEEE VLSI Test Symposium, 2008 (VTS'08)*, page 125–130, 2008.

[AKT+08] H. Ando, R. Kan, Y. Tosaka, K. Takahisa, and K. Hatanaka. Validation of hardware error recovery mechanisms for the SPARC64 v microprocessor. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, page 62–69, 2008.

[ALR01] A. Avizienis, J. C Laprie, and B. Randell. Fundamental concepts of dependability. Research report UCLA CSD Report no. 010028, 2001.

[AMD+10] M. Amin, F. Monteiro, C. Diou, A. Ramazani, and A. Dandache. A HW/SW mixed mechanism to improve the dependability of a stack processor. In *Proceedings of 16th IEEE International Conference on Electronics, Circuits, and Systems, 2009 (ICECS'09)*, page 976–979, 2010.

[ARM09] ARM. Cortex-R4 and Cortex-R4F. Technical reference manual, 2009.

[ARM+11] Mohsin Amin, Abbas Ramazani, Fabrice Monteiro, Camille Diou, and Abbas Dandache. A Self-Checking hardware journal for a fault tolerant processor architecture. *Hindawi Publishing Corporation*, 2011.

[Bai10] G Bailey. Comparison of GreenArrays chips with texas instruments MSP430F5xx as micropower controllers, June 2010.

[Bau05] R. C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3) :305–316, 2005.

[BBV+05] D. Bernick, B. Bruckert, P. D Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. In *Proceedings of International Conference on Dependable Systems and Networks, 2005 (DSN'05)*, page 12–21, 2005.

[BCT08] B. Bridgford, C. Carmichael, and C. W Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note*, 987, 2008.

[BGB+08] J. C Baraza, J. Gracia, S. Blanc, D. Gil, and P. J Gil. Enhancement of fault injection techniques based on the modification of VHDL code. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(6) :693–706, 2008.

[Bic10] R. Bickham. *An Analysis of Error Detection Techniques for Arithmetic Logic Units*. PhD thesis, Vanderbilt University, 2010.

[BP02] N. S Bowen and D. K Pradhan. Virtual checkpoints : Architecture and performance. *Computers, IEEE Transactions on*, 41(5) :516–525, 2002.

[BT02] D. Briere and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls-a family of fault-tolerant systems. In *the Twenty-Third International Symposium on Fault-Tolerant Computing System, 1993 (FTCS'93)*, page 616–623, 2002.

[Car01] C. Carmichael. Triple module redundancy design techniques for virtex FPGAs. *Xilinx Application Note XAPP197*, 1, 2001.

[Che08] L. Chen. Hsiao-Code check matrices and recursively balanced matrices. *Arxiv preprint arXiv :0803.1217*, 2008.

[CHL97] W-T Chang, S Ha, and E.A. Lee. Heterogeneous simulation - mixing Discrete-Event models with dataflow. *Journal of VLSI Signal Processing*, 15(1-2) :127–144, 1997.

[CP02] J. A Clark and D. K Pradhan. Fault injection : A method for validating computer-system dependability. *Computer*, 28(6) :47–56, 2002.

[CPB+06] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof : A defect-tolerant CMP switch architecture. In *Proceedings of 25th International Symposium on High-Performance Computer Architecture, 2006*, page 5–16, 2006.

[CTS+10] C. L. Chen, N. N. Tendolkar, A. J. Sutton, M. Y. Hsiao, and D. C. Bossen. Fault-tolerance design of the IBM enterprise system/9000 type 9021 processors. *IBM Journal of Research and Development*, 36(4) :765–779, 2010.

[EAWJ02] E. N. Elnozahy, L. Alvisi, Y. M Wang, and D. B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3) :375–408, 2002.

[EKD+05] D. Ernst, N. S Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, et al. Razor : A low-power pipeline based on circuit-level timing speculation. In *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003 (MICRO'03)*, page 7–18, 2005.

[FFMR09] R. Forsati, K. Faez, F. Moradi, and A. Rahbar. A fault tolerant method for residue arithmetic circuits. In *Proceedings of 2009 International Conference on Information Management and Engineering*, page 59–63, 2009.

[FGAD10] R. Fernández-Pascual, J. M Garcia, M. E Acacio, and J. Duato. Dealing with transient faults in the interconnection network of CMPs at the cache coherence level. *IEEE Transactions on Parallel and Distributed Systems*, 21(8) :1117–1131, 2010.

[FGAM10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring : probabilistic soft error reliability on the cheap. *ACM SIGPLAN Notices*, 45(3) :385–396, 2010.

[FP02] E. Fujiwara and D. K Pradhan. Error-control coding in computers. *Computer*, 23(7) :63–72, 2002.

[GBT05] S. Ghosh, S. Basu, and N. A Touba. Selecting error correcting codes to minimize power in memory checker circuits. *Journal of Low Power Electronics*, 1(1) :63–72, 2005.

[GC06] J. Gaisler and E. Catovic. Multi-Core processor based on LEON3-FT IP core (LEON3-FT-MP). In *in Proceedings of Data Systems in Aerospace, 2006 (DASIA'06)*, volume 630, page 76, 2006.

[Gha11] S. Ghaznavi. *Soft Error Resistant Design of the AES Cipher Using SRAM-based FPGA*. PhD thesis, University of Waterloo, 2011.

[GMT08] M. Grottke, R. Matias, and K. S Trivedi. The fundamentals of software aging. In *Proceedings of IEEE International Conference on Software Reliability Engineering Workshops, 2008 (ISSRE Wksp 2008)*, page 1–6, 2008.

[GPLL09] C. Godlewski, V. Pouget, D. Lewis, and M. Lisart. Electrical modeling of the effect of beam profile for pulsed laser fault injection. *Microelectronics Reliability*, 49(9-11) :1143–1147, 2009.

[Gre10] Green. Project green array chip, 2010.

[Hay05] J.R. Hayes. The architecture of the scalable configurable instrument processor. Technical Report SRI-05-030, The Johns Hopkins Applied Physics Laboratory, 2005.

[HCTS10] M. Y. Hsiao, W. C Carter, J. W Thomas, and W. R Stringfellow. Reliability, availability, and serviceability of IBM computer systems : A quarter century of progress. *IBM Journal of Research and Development*, 25(5) :453–468, 2010.

[HH06] A. J Harris and J. R Hayes. Functional programming on a Stack-Based embedded processor. 2006.

[Hsi10] M. Y Hsiao. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development*, 14(4) :395–401, 2010.

[IK03] R. K Iyer and Z. Kalbarczyk. Hardware and software error detection. Technical report, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana, 2003.

[Int09] Intel. White paper - the intel itanium processor 9300 series. Technical report, 2009.

[ITR07] ITRS. International technology roadmap for semiconductors. 2007.

[Jab09] Jaber. *Conception architecturale haut débit et sûre de fonctionnement pour les codes correcteurs d'erreurs*. PhD thesis, Université Paul Verlaine - Metz, France, Metz, 2009.

[Jal09] M. Jallouli. *Méthodologie de conception d'architectures de processeur sûres de fonctionnement pour les applications mécatroniques*. PhD thesis, Université Paul Verlaine - Metz, France, Metz, 2009.

[JDMD07] M. Jallouli, C. Diou, F. Monteiro, and A. Dandache. Stack processor architecture and development methods suitable for dependable applications. *Reconfigurable Communication-centric SoCs (ReCoSoC'07), Montpellier, France*, 2007.

[JES06] J. S JESD89A. *Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices*. October, 2006.

[JHW+08] J. Johnson, W. Howes, M. Wirthlin, D. L McMurtrey, M. Caffrey, P. Graham, and K. Morgan. Using duplication with compare for on-line error detection in FPGA-based designs. In *Proceedings of IEEE Aerospace Conference, 2008*, page 1–11, 2008.

[JPS08] B. Joshi, D. Pradhan, and J. Stiffler. Fault-Tolerant computing. 2008.

[KJ89] P. J Koopman Jr. *Stack computers : the new wave*. Halsted Press New York, NY, USA, 1989.

[KKB07] I. Koren, C. M Krishna, and Inc Books24x7. *Fault-tolerant systems*. Elsevier/Morgan Kaufmann, 2007.

[KKS+07] P. Kudva, J. Kellington, P. Sanda, R. McBeth, J. Schumann, and R. Kalla. Fault injection verification of IBM POWER6 soft error resilience. In *Architectural Support for Gigascale Integration (ASGI) Workshop*, 2007.

[KMSK09] J. W Kellington, R. McBeth, P. Sanda, and R. N Kalla. IBM POWER6 processor soft error tolerance analysis using proton irradiation. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic—Systems Effects (SELSE) Conference*, 2009.

[Kop04] H. Kopetz. *From a federated to an integrated architecture for dependable embedded systems*. PhD thesis, Technische Univ Vienna, Vienna, Austria, 2004.

[Kop11] H. Kopetz. *Real-time systems : design principles for distributed embedded applications*, volume 25. Springer-Verlag New York Inc, 2011.

[Lal05] P. K Lala. Single error correction and double error detecting coding scheme, 2005.

[Lap04] J.C. Laprie. Sûreté de fonctionnement des systèmes : concepts de base et terminologie. 2004.

[LAT07] K. W. Li, J. R. Armstrong, and J. G. Tront. An HDL simulation of the effects of single event upsets on microprocessor program flow. *IEEE Transactions on Nuclear Science*, 31(6) :1139–1144, 2007.

[LB07] J. Laprie and R. Brian. Origins and integration of the concepts. 2007.

[LBS+11] I. Lee, M. Basoglu, M. Sullivan, D. H Yoon, L. Kaplan, and M. Erez. Survey of error and fault detection mechanisms. 2011.

[LC08] C. A.L Lisboa and L. Carro. XOR-based low cost checkers for combinational logic. In *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, page 281–289, 2008.

[LN09] Dongwoo Lee and Jongwhoa Na. A novel simulation fault injection method for dependability analysis. *IEEE Design & Test of Computers*, 26(6) :50–61, December 2009.

[LRL04] J. C Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable Secure Computers*, 1(1) :11–33, 2004.

[MB07] N. Madan and R. Balasubramonian. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems*, page 1066–1079, 2007.

[MBS07] A. Meixner, M. E Bauer, and D. Sorin. Argus : Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, page 210–222, 2007.

[MG09] A. Maloney and A. Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation : Practice and Experience*, 21(12) :1632–1666, 2009.

[MM00] S. Mitra and E. J McCluskey. Which concurrent error detection scheme to choose ? 2000.

[MMPW07] K. S Morgan, D. L McMurtrey, B. H Pratt, and M. J Wirthlin. A comparison of TMR with alternative fault-tolerant design techniques for FPGAs. *IEEE Transactions on Nuclear Science*, 54(6) :2065–2072, 2007.

[Mon07] Y. Monnet. *Etude et modélisation de circuits résistants aux attaques non intrusives par injection de fautes*. Thèse de doctorat, Institut National Polytechnique de Grenoble, 2007.

[MS06] F. MacWilliams and N. Sloane. The theory of error-correcting codes. 2006.

[MS07] A. Meixner and D. J Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 104–118, 2007.

[MSSM10] M. J. Mack, W. M. Sauer, S. B. Swaney, and B. G. Mealey. IBM power6 reliability. *IBM Journal of Research and Development*, 51(6) :763–774, 2010.

[Muk08] S. Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2008.

[MW04] R. Mastipuram and E. C Wee. Soft errors' impact on system reliability. *EDN, Sept*, 30, 2004.

[MW07] T. C May and M. H Woods. A new physical mechanism for soft errors in dynamic memories. In *16th Annual Reliability Physics Symposium*, page 33–40, 2007.

[NBV+09] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L Scott. Fault injection framework for system resilience evaluation : fake faults for finding future failures. In *Proceedings of the 2009 workshop on Resiliency in high performance*, page 23–28, 2009.

[Nic02] M. Nicolaidis. Efficient implementations of self-checking adders and ALUs. In *Proceedings of Twenty-Third International Symposium on Fault-Tolerant Computing, (FTCS-23)*, page 586–595, 2002.

[Nic10] M. Nicolaidis. *Soft Errors in Modern Electronic Systems*. Springer Verlag, 2010.

[NL11] J. Na and D. Lee. Simulated fault injection using simulator modification technique. *ETRI Journal*, 33(1), 2011.

[NMGT06] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O : efficient handling of I/O in highly-available rollback-recovery servers. In *the Twety-fifth International Symposium on High-Performance Computer Architecture, 2006*, page 200–211, 2006.

[NTN+09] M. Nicolaidis, K. Torki, F. Natali, F. Belhaddad, and D. Alexandrescu. Implementation and validation of a low-cost single-event latchup mitigation scheme. In *IEEE Workshop on Silicon Errors in Logic–System Effects (SELSE), Stanford, CA*, 2009.

[NX06] V. Narayanan and Y. Xie. Reliability concerns in embedded system designs. *Computer*, 39(1) :118–120, 2006.

[Pat10] Anurag Patel. Fault tolerant features of modern processors, 2010.

[PB04] S Pelc and C. Bailey. Ubiquitous forth objects. In *Euro-forth'04*, Dahgstuhl, Germany, 2004.

[PF06] J. H Patel and L. Y Fung. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers*, 100(7) :589–595, 2006.

[Pie06] S.J. Piestrak. Dependable computing : Problems, techniques and their applications. In *First Winter School on Self-Organization in Embedded Systems*, Schloss Dagstuhl, Germany, 2006.

[Pie07] S.J. Piestrak. Systèmes numériques tolérants aux fautes, 2007.

[PIEP09] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3) :389–402, 2009.

[Poe05] Christian Poellabauer. Real-Time systems, 2005.

[Pow10] D. Powell. *A generic fault-tolerant architecture for real-time dependable systems*. Springer Publishing Company, Incorporated, 2010.

[QGK+06] H. Quinn, P. Graham, J. Krone, M. Caffrey, and S. Rezgui. Radiation-induced multi-bit upsets in SRAM-based FPGAs. *IEEE Transactions on Nuclear Science*, 52(6) :2455–2461, 2006.

[QLZ05] F. Qin, S. Lu, and Y. Zhou. SafeMem : exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture, 2005 (HPCA'11)*, page 291–302, 2005.

[RAM+09] A. Ramazani, M. Amin, F. Monteiro, C. Diou, and A. Dandache. A fault tolerant journalized stack processor architecture. In *15th IEEE International On-Line Testing Symposium, 2009 (IOLTS'09)*, Sesimbra-Lisbon, Portugal, 2009.

[RI08] G. A Reis III. *Software modulated fault tolerance*. PhD thesis, Princeton University, 2008.

[RK09] J. A Rivers and P. Kudva. Reliability challenges and system performance at the architecture level. *IEEE Design & Test of Computers*, 26(6) :62–73, 2009.

[RNS+05] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger. A smart card test environment using multi-level fault injection in SystemC. In *Proceedings of 6th IEEE Latin-American Test Workshop 2005*, page 103–108, March 2005.

[RR08] V. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *IEEE International Conference on Dependable Systems and Networks 2008 (DSN'08)*, page 1–10, Anchorage, Alaska, 2008.

[RRTV02] M. Rebaudengo, S. Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, page 210–218, 2002.

[RS09] B. Rahbaran and A. Steininger. Is asynchronous logic more robust than synchronous logic ? *IEEE Transactions on Dependable and Secure Computing*, page 282–294, 2009.

[RYKO11] W. Rao, C. Yang, R. Karri, and A. Orailoglu. Toward future systems with nanoscale devices : Overcoming the reliability challenge. *Computer*, 44(2) :46–53, 2011.

[Sch08] Martin Schoeberl. A java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 2008.

[SFRB05] V. Srinivasan, J. W. Farquharson, W. H. Robinson, and B. L. Bhuva. Evaluation of error detection strategies for an FPGA-Based Self-Checking arithmetic and logic unit. In *MAPLD International Conference*, 2005.

[SG10] L. Spainhower and T. A Gregg. IBM s/390 parallel enterprise server g5 fault tolerance : A historical perspective. *IBM Journal of Research and Development*, 43(5.6) :863–873, 2010.

[Sha06] Mark Shannon. *A C Compiler for Stack Machines*. MSc thesis, University of York, 2006.

[SHLR+09] S. K Sastry Hari, M. L Li, P. Ramachandran, B. Choi, and S. V Adve. mSWAT : low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 122–132, 2009.

[SMHW02] D. J Sorin, M. M.K Martin, M. D Hill, and D. A Wood. SafetyNet : improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th annual international symposium on Computer architecture*, page 123–134, 2002.

[SMR+07] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN'07*, page 297–306, 2007.

[Sor09] D.J. Sorin. Fault tolerant computer architecture, 2009.

[SSF+08] J. R Schwank, M. R Shaneyfelt, D. M Fleetwood, J. A Felix, P. E Dodd, P. Paillet, and V. Ferlet-Cavrois. Radiation effects in MOS oxides. *IEEE Transactions on Nuclear Science*, 55(4) :1833–1853, 2008.

[Sta06] William Stallings. *Computer Organization and Architecture*. Prentice Hall, 7th edition, 2006.

[TM95] C. H. Ting and C. H. Moore. Mup21 a high performance misc processor. *Forth Dimensions*, 1995.

[Too11] C. Toomey. *Statical Fault Injection and Analysis at the Register Transfer Level using the Verilog Procedural Interface*. PhD thesis, Vanderbilt University, 2011.

[Van08] V.P. Vanhauwaert. *Fault injection based dependability analysis in a FPGA-based enviroment*. PhD thesis, Institut Polytechnique de Grenoble, Gernoble, France, 2008.

[VFM06] A. Vahdatpour, M. Fazeli, and S. Miremadi. Transient error detection in embedded systems using reconfigurable components. In *International Symposium on Industrial Embedded Systems, 2006 (IES'06)*, page 1–6, 2006.

[VSL09] M. Vayrynen, V. Singh, and E. Larsson. Fault-tolerant average execution time optimization for general-purpose multi-processor system-on-chips. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2009 (DATE'09)*, page 484–489, 2009.

[WA08] F. Wang and V. D Agrawal. Single event upset : An embedded tutorial. In *21st International Conference on VLSI Design, 2008. VLSID 2008*, page 429–434, 2008.

[WCS08] P. M Wells, K. Chakraborty, and G. S Sohi. Adapting to intermittent faults in multicore systems. *ACM SIGPLAN Notices*, 43(3) :255–264, 2008.

[WL10] C. F. Webb and J. S. Liptay. A high-frequency custom CMOS s/390 microprocessor. *IBM Journal of Research and Development*, 41(4.5) :463–473, 2010.

[Yeh02] Y. C. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of IEEE Aerospace Applications Conference*, volume 1, page 293–307, 2002.

[ZJ08] Y. Zhang and J. Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual Reviews in Control*, 32(2) :229–252, 2008.

[ZL09] J. F. Ziegler and W. A. Lanford. The effect of sea level cosmic rays on electronic devices. *Journal of applied physics*, 52(6) :4305–4312, 2009.

RÉSUMÉ

Dans cette thèse, nous proposons une nouvelle approche pour la conception d'un processeur tolérant aux fautes. Celle-ci répond à plusieurs objectifs dont celui d'obtenir un niveau de protection élevé contre les erreurs transitoires et un compromis raisonnable entre performances temporelles et coût en surface. Le processeur résultant sera utilisé ultérieurement comme élément constitutif d'un système multiprocesseur sur puce (MPSoC) tolérant aux fautes. Les concepts mis en œuvre pour la tolérance aux fautes reposent sur l'emploi de techniques de détection concurrente d'erreurs et de recouvrement par réexécution. Les éléments centraux de la nouvelle architecture sont, un cœur de processeur à pile de données de type MISC (Minimal Instruction Set Computer) capable d'auto-détection d'erreurs, et un mécanisme matériel de journalisation chargé d'empêcher la propagation d'erreurs vers la mémoire centrale (supposée sûre) et de limiter l'impact du mécanisme de recouvrement sur les performances temporelles.
L'approche méthodologique mise en œuvre repose sur la modélisation et la simulation selon différents modes et niveaux d'abstraction, le développement d'outils logiciels dédiées, et le prototypage sur des technologies FPGA. Les résultats, obtenus sans recherche d'optimisation poussée, montrent clairement la pertinence de l'approche proposée, en offrant un bon compromis entre protection et performances. En effet, comme le montrent les multiples campagnes d'injection d'erreurs, le niveau de tolérance au fautes est élevé avec 100% des erreurs simples détectées et recouvrées et environ 60% et 78% des erreurs doubles et triples. Le taux recouvrement reste raisonnable pour des erreurs à multiplicité plus élevée, étant encore de 36% pour des erreurs de multiplicité 8.

Mots clés : Tolérance aux fautes, Processeur à pile de données, MPSoC, Journalisation, Restauration, Injection de fautes, Modélisation RTL.

ABSTRACT

In this thesis, we propose a new approach to designing a fault tolerant processor. The methodology is addressing several goals including high level of protection against transient faults along with reasonable performance and area overhead trade-offs. The resulting fault-tolerant processor will be used as a building block in a fault tolerant MPSoC (Multi-Processor System-on-Chip) architecture. The concepts being used to achieve fault tolerance are based on concurrent detection and rollback error recovery techniques. The core elements in this architecture are a stack processor core from the MISC (Minimal Instruction Set Computer) class and a hardware journal in charge of preventing error propagation to the main memory (supposedly dependable) and limiting the impact of the rollback mechanism on time performance.
The design methodology relies on modeling at different abstraction levels and simulating modes, developing dedicated software tools, and prototyping on FPGA technology. The results, obtained without seeking a thorough optimization, show clearly the relevance of the proposed approach, offering a good compromise in terms of protection and performance. Indeed, fault tolerance, as revealed by several error injection campaigns, prove to be high with 100% of errors being detected and recovered for single bit error patterns, and about 60% and 78% for double and triple bit error patterns, respectively. Furthermore, recovery rate is still acceptable for larger error patterns, with yet a recovery rate of 36%on 8 bit error patterns.

Keywords : Fault Tolerance, Stack Processor, MPSoC, Journalization, Rollback, Fault Injection, RTL modeling.