Architectures pour des circuits fiables de hautes performances

Thierry Bonnoit

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nano Electronique Nano Technologies**

Arrêté ministériel : 7 août 2006

Présentée par

Thierry BONNOIT

Thèse dirigée par **Michael NICOLAIDIS** et
codirigée par **Nacer-Eddine ZERGAINOH**

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

Architectures pour des circuits fiables de hautes performances

Thèse soutenue publiquement le **18 octobre 2012**
devant le jury composé de :

Monsieur, Patrick, GIRARD
Directeur de Recherche CNRS-LIRMM Montpellier, Président et Rapporteur
Monsieur, Luis, ENTRENA
Professeur des Universités, Universidad Carlos III de Madrid, Rapporteur
Monsieur, Michael, NICOLAIDIS
Directeur de Recherche CNRS-TIMA Grenoble, directeur de thèse
Monsieur, Nacer-Eddine, ZERGAINOH
Maitre de conférences TIMA Grenoble, co-directeur de thèse

Remerciements

Le travail présenté dans cette thèse a été effectué au sein de l'équipe ARIS du Laboratoire TIMA (CNRS–UJF–Grenoble INP), UMR 5159, de Grenoble. Je tiens à remercier sa directrice, Dominique Borrione, et tout le personnel administratif pour l'accueil chaleureux et l'assistance qu'ils m'ont prodigués. Je pense notamment à Anne-Laure Fourneret-Itié, Sophie Martineau, Lucie Torella et Laurence Ben Tito.

Je tiens à remercier également Monsieur Michael Nicolaidis, mon directeur de thèse, et Monsieur Nacer-Eddine Zergainoh, pour m'avoir encadré durant cette thèse. Je pense en particulier à la confiance qu'ils m'ont accordée durant ces quatre années en me confiant ce travail de recherche ; ils m'ont donné de précieux conseils et leurs suggestions ont grandement orienté ma réflexion.

Ma gratitude s'adresse également à Monsieur Patrick Girard, directeur de recherche au CNRS détaché au laboratoire LIRMM de l'université de Montpellier, pour l'honneur qu'il m'a fait en acceptant de réaliser le rapport de mon travail et la présidence de mon jury de thèse. Merci à Monsieur Luis Entrena, Professeur à l'Universidad Carlos III de Madrid, également rapporteur cette thèse.

Enfin, j'aimerais remercier Monsieur Alexandre Chagoya pour son assistance informatique au CIME, et bien entendu l'ensemble du service informatique de TIMA, Frédéric Chevrot, Nicolas Garnier et Ahmed Khalid.

Je veux également dire merci à tous ceux qui ont partagé ces quatre années au laboratoire, Fabien Chaix, Vladimir Pasca, Panagiota Papavramidou, Salma et Mohamed Ben Jrad, Diarga Fall et son frère Ibou-Tine Fall que j'ai eu le plaisir d'encadrer durant son stage à TIMA, Gilles Bizot, Adrien Prost Boucle, Adrian Evans, Maryam Bahmani...

Mes plus sincères remerciements vont également à Monsieur et Madame Jean-Claude et Jacqueline Sabonnadière, pour leur disponibilité, leurs encouragements, leurs conseils et leur gentillesse.

Je dédie spécialement ce travail à ma famille, à commencer par mes parents Jean et Mireille qui comme tant de retraités sont finalement beaucoup trop occupés, à mon frère Pierre et sa compagne Hélène ainsi qu'à mon neveu, Julien, qui vient d'arriver, à ma sœur Magali et son compagnon Damien en souhaitant qu'ils voyagent moins loin. Je dédie aussi cet ouvrage à tous mes oncles et tantes, cousins et cousines, parrain et marraine, trop nombreux pour être cités ici, que j'ai si peu vu durant ces années grenobloises et qui ont si souvent demandé de mes nouvelles.

Enfin, et alors qu'il y aurait encore tant de gens à remercier, et à qui je demande pardon de ne pas les citer, je ne peux m'empêcher au moment de conclure ce propos liminaire d'avoir une pensée pour Madame Jacqueline Millet, et son ô combien regretté époux Monsieur Bernard Millet, qui du haut de ses 85 ans m'avait fait l'honneur de son amitié.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

La fabrication des circuits intégrés modernes est grandement influencée par l'arrivée des technologies nanométriques dans la conception des transistors CMOS (Complementary Metal-Oxide-Semiconductor). Les premières conséquences, les plus intéressantes pour les fabricants, sont la capacité de pouvoir intégrer de plus en plus d'éléments dans une même puce électronique, tout en augmentant la fréquence et en limitant la consommation énergétique. La contrepartie de cette tendance est une diminution significative de la fiabilité des circuits. Cette perte est due d'une part aux défauts pouvant apparaître lors de la fabrication de ces circuits, d'autre part du fait de leur sensibilité accrue aux phénomènes extérieurs.

Le monde de la micro-électronique s'est efforcé, et s'efforce encore aujourd'hui, de suivre la loi de Moore. En effet, le nombre de transistors pouvant être intégré sur une même unité de surface double quasiment tous les deux ans [Int05]. Au regard des investissements, et des efforts consentis pour retarder les effets dus aux limites physiques des technologies CMOS, il est raisonnable de penser que cette tendance va se poursuivre au cours de la prochaine décennie. Cette évolution est favorisée en outre par le nombre incalculable d'applications concernées par la micro-électronique dans les différents domaines de la vidéo, des télécommunications, de la production industrielle, de l'automobile, de la sécurité, de l'aérospatiale, sans oublier la médecine et le nucléaire… Cette tendance va contribuer à accroître encore la complexité des puces produites (Fig 1) [ITRS09].

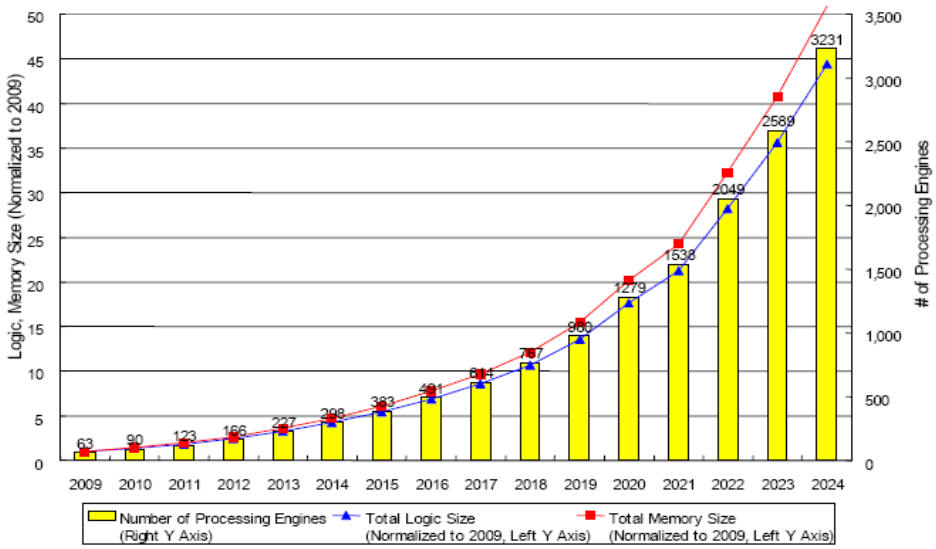

Fig. 1 : Evolution de la complexité des circuits (ITRS 2009)

En raison de cet accroissement de complexité, couplée avec la diminution de la dimension des transistors et l'augmentation de la fréquence de fonctionnement, les fabricants doivent faire face à une réduction de la fiabilité des circuits électronique. Cette réduction est due notamment à une augmentation des erreurs transitoires.

La fiabilité des circuits est perturbée par le fait que la dimension d'un transistor est devenue comparable à la taille des atomes (entre 0,1 et 0,2 nm). Les perturbations extérieures telles que les flux de neutrons ou les radiations alpha ont donc un impact plus significatif sur les composants électroniques [Bau05] [BTN+09] [Met09]. Celles-ci proviennent de rayonnements cosmiques qui, en traversant notre atmosphère, se décomposent en particules secondaires dont des neutrons électriquement neutre, mais qui vont interagir avec les atomes de la puce, et ainsi engendrer des impulsions électriques parasites. Dans le cas d'environnements plus critiques, spatiale ou fortement ionisants comme le nucléaire, les problèmes dus entre autre aux radiations sont encore plus importants, alors que les critères de sécurité et de fiabilité sont beaucoup plus élevés.

Cela peut se traduire par une modification de l'information contenue dans un composant de stockage, ou par une perturbation du résultat produit par une fonction logique. La réduction de la tension d'alimentation conduit à une réduction de la marge d'erreur des composants, et donc accroit la probabilité qu'une perturbation vienne changer l'état d'un composant. Pour un composant de stockage, telle qu'une mémoire RAM, qui est conçue de façon aussi dense que possible, cela accroît l'étendue des perturbations. Pour les modules de calculs, l'accroissement de la fréquence de fonctionnement du circuit, et l'augmentation de leur complexité font qu'une perturbation dans le résultat d'une fonction logique a plus de chance d'être capturée au final par un composant de stockage (bascule, registre, mémoire). Enfin, les différences de fonctionnement dues à la variabilité dans le processus de fabrication, font que la sensibilité aux phénomènes extérieurs varie d'un circuit à un autre [TEI+04] [RPS+05] [ITY+10].

Au total, les erreurs transitoires peuvent se classer en trois catégories.

- Les fautes de délais : le résultat d'une fonction logique est correct, mais avec un retard sur le délai prévu. Ce retard peut faire en sorte que le résultat correct soit produit après la transition d'horloge. Ces fautes sont liées en général aux problèmes de variabilités.
- Les évènements transitoires ou Single Event Transient (SET) : une ou plusieurs fonctions logiques sont perturbées par un phénomène extérieur qui résulte en la propagation d'une impulsion parasite à travers le circuit. Comme la faute de délai, elle conduit à une erreur transitoire si ces impulsions sont capturées lors d'une transition d'horloge.
- Les changements d'état des composants de stockage ou Single Event Upset (SEU) : un ou plusieurs bits stockés dans un composant de stockage ont été modifiés à la suite d'une perturbation extérieure. Ces changements d'état peuvent également se différentier en plusieurs catégories :
  - un seul évènement provocant le changement d'état d'un seul bit ou Single Bit Upset (SBU) ;

- un seul évènement provocant le changement d'état de plusieurs bits en même temps ou Multiple Cells Upset (MCU) ;
- un seul évènement provocant le changement d'état de plusieurs bits en même temps d'un même mot enregistré en mémoire ou Multiple Bits Upset (MBU).

Il y a deux manières de résoudre ces problèmes. La première consiste à créer des composants électroniques plus résistants aux radiations, mais également plus coûteux. La seconde, qui peut se situer aussi bien au niveau de l'architecture du circuit qu'au niveau logiciel, consiste à détecter et corriger ces erreurs dues à des phénomènes transitoires. C'est ce dernier axe qui est concerné par les travaux de recherche suivants.

Ce problème de fiabilité est plus préoccupant pour les mémoires, car elles occupent la plus grande partie du circuit, et concentrent le plus grand nombre de transistors. Elles sont plus sensibles aux perturbations extérieures car elles sont conçues de manière aussi dense que possible, dans les limites autorisées par le processus de fabrication. Cela conduit à un accroissement du taux de MCU, et donc potentiellement du taux de MBU [RPS+05] [ITY+10]. Les techniques dédiées à la réduction de la probabilité d'erreurs pour les mémoires RAM peuvent se regrouper pour l'essentiel en trois catégories : détection instantanée des SEU (capteurs de courant intégrés à la mémoire), architectures des mémoires (design des cellules mémoires SRAM renforcées), et redondance des données (redondance modulaire, code correcteurs d'erreurs). Certaines solutions peuvent être combinées ensemble et/ou complétées par le multiplexage de colonnes qui consiste à éloigner le plus possible les bits appartenant à un mot enregistré en mémoire. Ce faisant, cela réduit la possibilité qu'une erreur transitoire affecte plus d'un bit à la fois dans un même mot stocké en mémoire.

Les codes correcteurs d'erreurs constituent l'une des solutions les plus utilisées aujourd'hui car ils permettent d'assurer une bonne fiabilité à un coût modéré. Toutefois, l'augmentation de la sensibilité des mémoires embarquées conduit à utiliser des codes plus complexes qu'avant. Dans le même temps, l'augmentation de la fréquence de fonctionnement des circuits fait que l'utilisation de ces codes a une influence négative sur la bande passante du système. En effet les circuits de codage et de décodage induisent une pénalité de temps qui peut être inacceptable dans un certain nombre de circuits.

L'objectif de ce travail est de proposer un système de protection à base de code correcteur qui soit compatible avec les nouvelles tendances technologiques, que ce soit en termes de fiabilité ou de vitesse de fonctionnement. Pour cela nous proposons une méthode au niveau architectural permettant l'utilisation des codes les plus complexes sans abaisser les performances du système. De plus, la solution proposée doit limiter le surcoût matériel du à la méthode et être facilement intégrable dans un circuit complexe.

Contribution de la thèse

Nous avons proposé une méthode d'implémentation des codes correcteurs au niveau RTL qui peut s'adapter à n'importe quel type d'architecture synchrone. Cette méthode supprime la pénalité de temps due à l'utilisation de ces codes lors de l'écriture des données en mémoire. De plus elle limite la pénalité de temps aux seuls cas où une erreur est détectée lors de la lecture d'une donnée en mémoire. Globalement, la pénalité de temps due aux codes correcteurs est donc limitée aux seuls cas où une erreur est effectivement détectée. Cette technique permet donc de limiter les conséquences sur la bande passante du système, dues à l'utilisation de codes correcteurs, avec un surcoût matériel par rapport à une implémentation classique des codes correcteurs relativement faible.

Pour ce faire on procède à la décontamination du circuit après qu'une donnée erronée ait été propagée dans le circuit ; le contrôle de l'intégrité des données se faisant en parallèle du fonctionnement normal du circuit. Cette décontamination nécessite la restauration de certains des états précédents de quelques composants de stockage du circuit. Cette restauration est réalisée par l'ajout de FIFO qui vont préserver les données précédentes de ces composants de stockage pendant quelques cycles d'horloge. Cependant, la complexité de beaucoup de circuits, tant au niveau de l'architecture des modules de calcul que du nombre de mémoires utilisées, rend difficile une implémentation manuelle de cette solution. C'est pourquoi, grâce à une représentation abstraite d'un circuit, un algorithme permettant d'identifier les composants du circuit dont les états doivent être préservés a également été créé. Nous avons ensuite essayé d'évaluer l'impact qu'aurait l'utilisation de cette méthode dans le contexte plus large suivant : la restauration d'un état précédent de l'ensemble du circuit en vue de corriger une erreur susceptible de se produire n'importe où dans le circuit. Au final, ce rapport de thèse est composé de quatre chapitres :

Le chapitre 1 fait le point sur les problèmes de fiabilité des circuits électroniques face aux erreurs transitoires. En particulier les tendances actuelles y sont détaillées, ainsi que les différentes techniques au niveau architectural qui sont proposées pour contrecarrer ces phénomènes.

Le chapitre 2 est consacré à l'exposé de la méthode permettant de supprimer lors de l'écriture des données en mémoire le délai dû aux codes correcteurs, et de le limiter lors de la lecture d'une donnée aux seuls cas où une erreur doit être corrigée. L'architecture détaillée de la solution y est exposée pas à pas pour un circuit simple. Enfin sont détaillés les ajustements à effectuer pour implémenter cette solution dans les architectures les plus complexes. L'importance des composants de stockage dont on doit restaurer les états précédents pour effectuer une décontamination correcte est mise en évidence.

Le chapitre 3 traitera de la modélisation théorique et de l'algorithme utilisé pour implémenter la technique présentée dans le chapitre 2. Le circuit est modélisé sous la forme d'un graphe orienté. Ce modèle permet d'établir les propriétés que vérifient les composants de stockage dont les états précédents doivent être préservés pour la décontamination. A partir de cette étude, l'algorithme permettant l'identification de ces composants, ainsi qu'une méthode pour réduire le coût matériel dû à l'utilisation de cette technique sont détaillés

Le chapitre 4 traite de la restauration d'un état précédent quelconque de l'ensemble du circuit, comme extension de la technique précédente. Ce chapitre comporte une étude comparative de plusieurs solutions proposées. Ces solutions sont basées sur le même principe de préservation des états précédents du circuit en minimisant le coût matériel. Les modifications à apporter à l'algorithme présenté au chapitre 3 pour résoudre ce nouveau problème y sont également présentées.

Les chapitres 2, 3 et 4 traitant de problématiques connexes, et en même temps relativement différentes, chacun possède sa section dédiée aux résultats expérimentaux. Le mémoire se conclura par un bilan des travaux effectués ainsi que des perspectives futures que l'on peut en attendre.

Chapitre 1. Vulnérabilité et protection des circuits face aux erreurs transitoires

---

Les technologies nanométriques font face à des problèmes de fiabilités qui sont dues notamment à un accroissement des erreurs transitoires. Ces erreurs ont pour origines le fait que les circuits actuels sont plus sensibles aux radiations. Les mémoires sont les composants les plus concernées par ces phénomènes, car elles occupent souvent la plus grande partie des circuits. Le problème le plus préoccupant pour les mémoires à l'heure actuelle est l'augmentation des erreurs multiples provoquées par une seule particule. La fiabilité des circuits logiques est également mise en cause, car les impulsions parasites crées par les radiations voient leur probabilité d'être capturée par un composant de stockage augmenter. Ce chapitre fait le point sur les tendances actuelles concernant la résistance des circuits aux erreurs transitoires, ainsi que les techniques architecturales développées pour détecter, masquer et corriger ces erreurs.

1.1 Introduction

La fiabilité des circuits intégrés modernes est affectée par les dimensions nanométriques des transistors. Une conséquence de cette tendance, est la nécessité de trouver de nouvelle approche pour améliorer la fiabilité des circuits [BTN+09] [LYL09] [Met09]. La prise en compte de la diminution de cette fiabilité s'est faite progressivement, au fur et à mesure des défaillances constatées. Parmi les causes de cette baisse de fiabilité, plusieurs mécanismes, ayant pour origines des radiations, provoquent des erreurs transitoires. Une donnée déjà stockée dans un composant de type mémoire, ou une donnée utilisée dans une fonction logique dont le résultat pourrait être stocké par la suite, sera modifiée de façon temporaire. Les mots "transitoire" et "temporaire" doivent ici être compris dans le sens où les zones affectées peuvent être réutilisées. On peut en effet réécrire une autre donnée dans un composant de stockage, ou bien le phénomène affectant une fonction logique se dissipe au bout d'un certains temps. Ces sources d'erreurs ne sont pas les mêmes suivants les milieux considérés, en particulier si on est au niveau de la mer, en altitude, ou dans l'espace.

1.1.1 Historique de la prise en compte des erreurs transitoires

Si depuis 1962 [WM62] l'apparition des erreurs transitoires avait été prévue, les premiers constats dans le domaine de l'aérospatiale datent de 1975 [BSH75]. Ces défaillances concernaient plusieurs bascules des circuits, à hauteur de quatre "anomalies" en 17 ans pour des circuits équipant un satellite. Depuis, la constatation des effets dus aux radiations s'est faite en parallèle de la découverte et de l'étude des mécanismes qui ont conduit à la création de ces erreurs transitoires. Ce qui suit est un résumé succinct des étapes les plus marquantes dans la prise en compte de ces erreurs.

La première publication scientifique sur les mécanismes physique à l'origine des erreurs transitoires date de 1978 [MW78] par Intel. Ces erreurs transitoires sont définies comme étant des évènements non permanents, aléatoires, qui ne sont pas causés par du bruit sur des signaux électriques ou des interférences électromagnétiques, mais causés par des radiations. Toujours en 1978 Ziegler et Lanford d'IBM démontrèrent que les rayonnements cosmiques interagissaient avec l'atmosphère terrestre, créant ainsi plusieurs particules secondaires (protons, électrons, neutrons…), qui pouvaient ensuite réagir avec le silicium [ZL79]. Les premiers résultats concernant l'apparition des erreurs transitoires au niveau de la mer datent de 1983 par O'Gorman [Gor94]. Après trois ans de mesures en sous-sol, au niveau de la mer, et en altitude, les résultats ont montré que des radiations atmosphériques (alpha et neutrons) étaient à l'origine de plusieurs erreurs transitoires. En 1993 il a été montré qu'une mémoire SRAM de 256 kbits incluse dans le système informatique d'un avion de ligne commercial, avait un taux de défaillance de 1 tous les 80 jours par puce [OBF+93]. Il a été constaté en 1996, qu'une mémoire DRAM de 156 Gbits d'un superordinateur pouvait être affectée par des erreurs transitoires plusieurs fois par jour [Nor96]. Des problèmes de tolérance aux fautes transitoires ont été également constatés pour des pacemakers [BN98] de la même façon que s'ils étaient impactés par un

flux de neutrons. En 1999 Sun Microsystems a connu de graves problèmes qui ont résulté en la perte d'un grand nombre de consommateurs. En cause la réinitialisation de leur serveur "Entreprise" plusieurs fois en quelques mois sans raison apparente [For00]. Enfin Cisco en 2003 a eu le même genre de problème sur leur carte de routage (12000 series) [Cis03]. L'erreur, ayant pour origine des radiations, était détectée par un système de code correcteur et provoquait ainsi la réinitialisation du système, laquelle durait 3 à 4 minutes.

1.1.2 Les phénomènes physiques à l'origine des erreurs

Depuis l'étude des erreurs transitoires, et des phénomènes physiques qui sont à l'origine de celles-ci, trois mécanismes ont été identifiés. Aujourd'hui deux de ces trois mécanismes sont devenus négligeables par rapport au troisième. Ces mécanismes ont pour conséquence la création d'ions à l'intérieur de la puce, ce qui induit une impulsion électrique parasite.

Vers la fin des années 1970, début des années 1980, des particules alpha émises par des impuretés d'uranium et de thorium radioactifs, ont été détectées dans les puces électroniques. Ces particules ont été identifiées comme la source d'erreur principale des mémoires DRAM de l'époque au niveau terrestre. Cette source d'erreur est aujourd'hui minoritaire grâce à l'utilisation de matériaux extrêmement purifiés [Bau05].

A peu près à la même époque, les neutrons très énergétiques (plus de 1 MeV - Mega électron Volt), ayant pour origine les radiations cosmiques qui engendrent des particules secondaires dans l'atmosphère terrestre, ont été identifiés comme une source indirecte d'erreurs transitoires. Si les particules alpha sont électriquement chargées et créent un ensemble de paires électron-trous quand elles traversent un circuit intégré, les neutrons sont électriquement neutres. Mais un neutron d'énergie élevée peut interagir avec du silicium, de l'oxygène ou tout autre atome de la puce, pour créer des particules secondaires électriquement chargées capables d'induire des erreurs transitoires. L'amplitude de ces phénomènes est caractérisée par le *Linear Energy Transfer* (LET) d'un ion, exprimé en $MeV.cm^2/mg$. Plus les ions créés sont lourds et très énergétiques dans des circuits où la densité de transistors est très grande, plus ceux-ci causent des perturbations importantes. C'est le cas des ions de magnésium, un des ions couramment produits par l'interaction des neutrons avec les atomes de silicium. Il s'agit aujourd'hui de la source d'erreurs la plus courante au niveau terrestre [Bau05]. Cela est illustré par la figure suivante (Fig.1-1) qui indique le flux de neutrons reçu au niveau terrestre par unité de surface et par seconde, en fonction de leur énergie [GGR+04] et la conséquence de ce flux sur la création d'ion dans le silicium [ITY+10] en unité arbitraire (A.U.). Aujourd'hui la sensibilité des circuits est arrivée à un point où même les neutrons de plus faibles énergies ne sont plus à négliger [ITY+10]. L'influence modératrice de l'atmosphère s'amenuise avec l'altitude, et la situation est encore plus critique dans l'espace. Ainsi les radiations solaires sont constituées de rayonnements alpha, beta et d'ions lourds, d'énergie atteignant le giga électron Volt. Les radiations cosmiques constituées de rayonnements alpha et d'ions lourds peuvent atteindre des énergies jusqu'au tera électron Volt [BDS03].

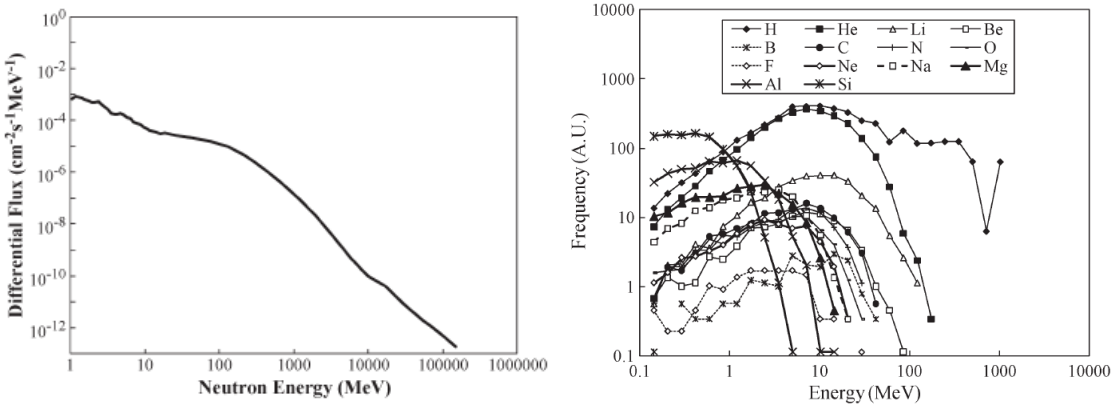

Fig. 1-1 : Flux de neutrons reçus au niveau terrestre [GGR+04] (à gauche), et ses conséquences sur la création d'ions dans le substrat silicium [ITY+10] (à droite).

Enfin les neutrons thermiques ont été également identifiés comme une source importante d'erreurs transitoires. Cette source découverte en 1995, était due à la présence d'un isotope particulier de bore dans la fabrication des composants (verre de borophosphosilicate - BPSG), et son effet pouvait être activé par des neutrons de faible énergie. Cette source a été éliminée par un changement dans les processus de fabrication en remplaçant le BPSG par un matériau diélectrique [BHM+95] [Bau05].

Quand un nœud sensible, souvent le drain d'un transistor, est près de la zone d'ionisation d'une particule électriquement chargée (Fig. 1-2 a), il rassemble une partie significative des porteurs de charges générés (des trous ou des électrons, Fig. 1-2 b), avant que l'état du nœud ne se stabilise (Fig. 1-2 c). Le résultat est une impulsion électrique parasite sur ce nœud (chronogramme de la figure 1-2). L'effet de cette impulsion dépend du type de composant auquel le nœud appartient [DNR02].

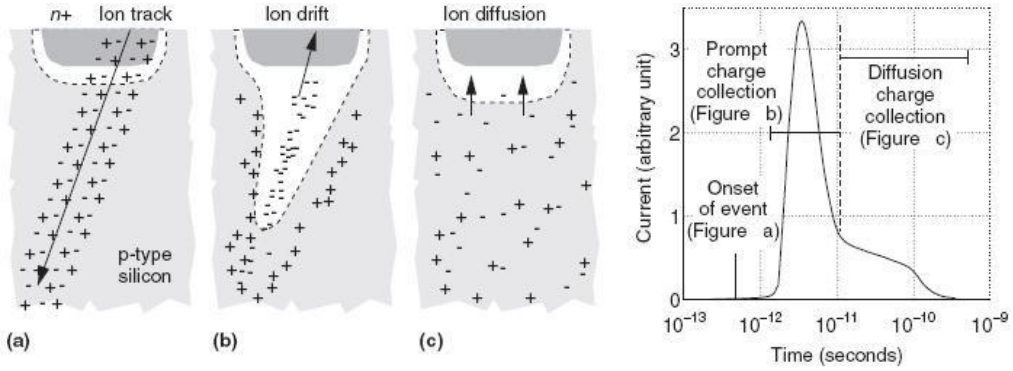

Fig. 1-2 : Création d'une impulsion de courant au niveau du nœud d'un transistor [Bau05]

Dans un composant de stockage (une mémoire, une bascule…) une impulsion parasite suffisamment forte changera l'état du composant. Quand le nœud appartient à une porte logique, l'impulsion électrique est transformée en une impulsion de tension (*Single Event Transient*, SET), dont la forme et l'amplitude dépendent des caractéristiques de l'impulsion et des caractéristiques électriques du nœud. L'impulsion parasite peut être propagée jusqu'à un composant de stockage, modifiant ainsi le résultat final si elle coïncide avec une transition d'horloge. Dans ces deux cas, le résultat est un

Single Event Upset (SEU), c'est-à-dire une erreur temporaire ayant modifié l'état du circuit. Le cas le plus typique des erreurs transitoires se situe au niveau des mémoires RAM, à cause de leur importante densité, et de la surface qu'elles occupent dans bon nombre d'architectures [Nic05]. Les composants électroniques peuvent également être perturbés par des erreurs permanentes. C'est-à-dire que l'état d'un nœud est irrémédiablement fixé (collé) à '1' ou '0', ou encore collé à un autre nœud. Dans la suite de ce mémoire nous ne parlerons que des erreurs transitoires.

1.2 Evolution des technologies mémoires et perspectives

Les erreurs transitoires sont plus préoccupantes pour les mémoires, car ces dernières occupent la plus grande partie des circuits, et concentrent la plus importante partie des transistors. Elles sont plus sensibles aux défaillances car elles sont conçues de manière aussi dense que possible, dans les limites autorisées par le processus de fabrication. La complexité du circuit, la densité des mémoires, la réduction de la tension d'alimentation, l'augmentation de la fréquence de fonctionnement, et le processus de fabrication influent grandement sur leur capacité à résister aux SEU [TEI+04] [KHN+04] [RPS+05] [ICS+06]. A titre d'exemple une réduction de la tension d'alimentation de 1.25V à 0.7V pour une même mémoire SRAM double la valeur du taux de SEU, appelé *Soft Error Rate* (SER), et le double encore quand on réduit la tension de 0.7V à 0.5V, ce qui peut causer de sérieux problèmes pour des systèmes utilisant le DVFS (*Dynamic Voltage Frequency Scaling*), une technique utilisée pour réduire la consommation d'énergie dans les microprocesseurs [DW11]. Toutefois il est difficile de prévoir l'évolution de leur sensibilité aux erreurs transitoires. Nous allons faire le point sur la sensibilité des cellules mémoires actuelles et, en particulier, l'augmentation pour les mémoires SRAM du taux d'erreurs multiples.

1.2.1 Evolution des mémoires DRAM et SRAM

Il était prévu que les DRAM, conçues pour être des mémoires très denses, verraient leur taux de défaillance continuer à augmenter. A l'inverse c'est l'un des composants les plus robustes aux SEU actuellement. Comme illustré par la figure 1-3 ([Bau05]) le SER par bit pour les DRAM a incroyablement chuté, plus de 1000 fois moins. Ceci est du à un changement dans la fabrication des cellules de base. Dessinées de façon planaire au départ, elles collectaient aisément les charges créées par les radiations atmosphériques. La mise au point d'une cellule DRAM dessinée en 3D a eu pour effet de les rendre moins sensibles aux radiations. Mais cette tendance est compensée par l'augmentation de la densité de ces mémoires, ce qui est permis par la réduction des dimensions. On implémente ainsi des mémoires embarquées aux capacités de stockage toujours plus importantes. De fait, les systèmes à base de DRAM restent avec un SER global quasiment constant. A l'inverse les SRAM ont vu leurs sensibilités accrues, alors que leur architecture à l'origine devait forcer l'état des transistors à rester constant. Mais avec la diminution des dimensions, la charge critique Q_{crit} nécessaire pour changer l'état

de l'un des transistors d'une cellule a diminué (Fig. 1-3). La plupart des technologies, que nous citerons plus loin, destinées à renforcer les cellules des mémoires SRAM, sont basées sur la réduction de la sensibilité des cellules, en rehaussant la valeur de la charge critique des transistors qui les composent. Enfin [SSK+06] et [DW11] affirment que le SER par bit des mémoires SRAM est en train de se réduire, mais pas encore suffisamment pour compenser l'augmentation de la densité des mémoires. Ainsi [ITY+09] évalue, entre les technologies 130 nm et 22 nm, une diminution de la sensibilité par bit d'un rapport 4, pour un SER total finalement multiplié par 7.

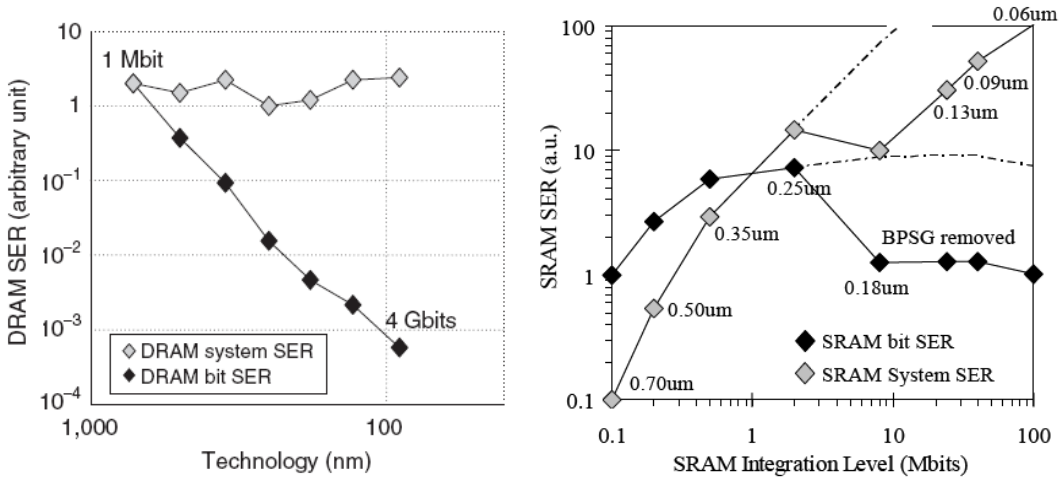

Fig. 1-3 : SER, par bit et par puce, des mémoires DRAM (à gauche) et SRAM (à droite), en fonction de la technologie ([Bau05])

Globalement les prévisions restent pessimistes. Par exemple l'ITRS (*International Technology Roadmap for Semiconductors*) a fixé comme objectif un SER de 1150 FIT/Mbit (1 FIT = 1 erreur toute les 10^9 heures) pour les mémoires SRAM avec une technologie 65 nm. L'ITRS prévoit une évolution pour des technologies d'une dizaine de nm jusqu'à plus de 1400 FIT par Mbit, malgré les tendances, signalées plus haut, de voir diminuer le SER par bit des mémoires SRAM [ITRS09]. A titre de comparaison Ibe (Hitachi) fixe un objectif plus ambitieux ; limiter le SER à 200-400 FIT par Mbit [Ibe08], soit 3 à 4 fois moins que les tendances actuelles prévues.

1.2.2 Les erreurs multiples dans les mémoires SRAM

Un point important soulevé par l'ITRS est le nombre de bits stockés en mémoire, affectés en même temps, par un seul SEU. Selon l'ITRS, les mémoires embarquées évoluent vers un point où tout SEU conduira à avoir plusieurs cellules mémoires affectées simultanément, sans toutefois présumer de l'étendue de l'impact. En effet, les SEU peuvent affecter un seul bit (*Single Bit Upset*, SBU), ou plusieurs bits. Toutefois il n'est pas garanti, lorsque plusieurs bits sont affectés, que ceux-ci appartiennent à une même donnée stockée en mémoire. Par conséquent on parlera de MCU (*Multi Cells Upset*) lorsque plusieurs bits sont affectés, sans distinguer le nombre de bits affectés pour chacune des données corrompues. On parlera de MBU (*Multi Bits*

Upset) dans le cas où de façon certaine au moins une donnée stockée a vu plus d'un bit modifié. Dans ce dernier cas, les erreurs sont plus difficiles à corriger.

Les résultats concernant les MCU dans la littérature sont assez variables. Dans [SSW09] les MCU sont évalués entre 20 et 40% des SEU totaux pour les technologies SRAM 65 et 45 nm. A l'inverse [Ale11] indique pour une mémoire SRAM 45 nm que les MCU ne représenteraient que 27% des erreurs. Toutefois, globalement les résultats sont plus optimistes que les objectifs fixés par l'ITRS. Selon [ITY+10], pour les technologies SRAM 32 nm, les MCU sont estimés à 38% des SEU, contre 64% prévus par l'ITRS [ITRS09], et à 48% pour une mémoire SRAM dans une technologie 22 nm contre 100% par l'ITRS. Quoiqu'il en soit, la proportion des MCU se rapproche des 50%. Ce qui signifie qu'une erreur transitoire sur deux affecte plusieurs bits en mémoire. Pour [Ale11], les résultats présentés ont été fournis à l'aide de l'outil TFIT qui simule l'effet électrique sur le circuit d'une particule incidente [BPN+06] [TNU+09]. La très grande majorité des MCU affecte alors des cellules mémoires très proches (Fig. 1-4).

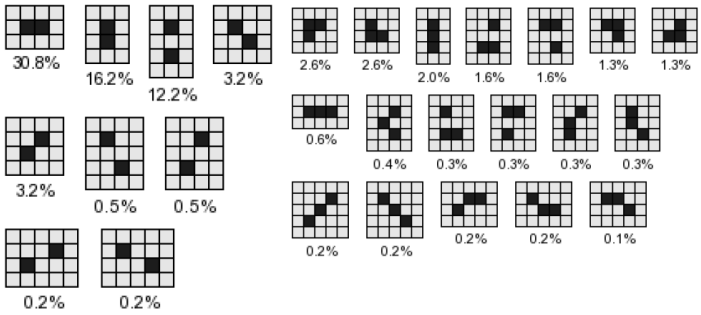

Fig. 1-4 : Emplacement des bits faux dans les erreurs doubles et triples ([Ale11])

Toujours dans [Ale11] il est affirmé que, pour une mémoire SRAM fabriquée dans une technologie 45 nm, les MCU affectent en moyenne plus de 3 bits par SEU. Dans [ITY+10], il est prédit que plus d'une dizaine de cellules mémoire sur une même ligne (donc appartenant potentiellement au même mot mémoire) peuvent être corrompues en même temps pour les technologies 32 et 22 nm. Cela n'arrive toutefois qu'avec de très faibles probabilités, les MCU affectant plusieurs bits sur une même ligne n'étant évalués qu'à 3 et 4% des SEU totaux. A noter la différence surprenante avec les MCU affectant plusieurs bits en colonne : 45% des SEU avec des multiplicités supérieures à 100. Ce qui tendrait à dire que des MBU dues à plusieurs SBU deviennent plus probable, les MCU affectant un grand nombre de lignes en même temps. Les tests dans [ITY+10] ont été faits en utilisant un protocole basé sur le modèle de flux d'électrons au niveau terrestre présenté en début de chapitre [JED89].

Dans [RPS+05] les tests physiques ont été réalisés sur une mémoire SRAM dans une technologie 150 nm avec des particules incidentes dont les énergies étaient comprises entre 22 MeV et 144 MeV. Les résultats montrent une proportion de MCU variant entre 30 et 40%. Si on ne considère que les MCU de deux bits, ceux-là comptent pour plus de 20% des SEU, et impliquent dans plus de 70% des cas des MBU de deux bits (sans technique pour les éviter). Les MCU de 3 bits représentent de 3 à 13% des SEU. En

revanche la proportion des MBU de trois bits engendrés n'est pas donnée avec précision, mais représente moins de 3 à 6% des MCU. Les MCU de 4 bits représentent de 2 à 7% des SEU, et engendrent des MCU supérieurs à 2 bits dans une proportion variant de moins de 7% à moins de 14% des MCU.

D'autres résultats montrent que l'impact d'une particule très énergétique, venant perturber le circuit, peut engendrer des phénomènes physiques différents suivant l'orientation de l'impact [MOR+96] [MSN+00] [GNP05]. Dans [CMF+98], les tests sur une technologie SRAM 0.8µm ont démontré que des cellules mémoires peuvent être affectées par un même phénomène bien que séparées par des distances allant jusqu'à :

- 300 µm d'écart en colonne (ce qui représente 12 cellules mémoire dans la technologie considérée),
- 88 µm en ligne (soit 5 cellules mémoire).

Ces résultats ont été obtenus en considérant des ions lourd de magnésium à incidence rasante (88° / verticale) d'énergie supérieure à 1GeV.

1.2.3 Le multiplexage de colonne

Parmi les techniques existant pour renforcer les mémoires, il y a le multiplexage de colonne. Le principe est basé sur l'idée que les cellules affectées par un MCU sont proches, et qu'il suffit d'éloigner les bits appartenant à un même mot enregistré en mémoire pour éviter les MBU. Aussi certaines publications parlent du facteur de multiplexage nécessaire (nombre de bits intercalés entre deux bits appartenant au même mot, sur une même ligne mémoire) pour éviter l'apparition des MBU (Fig. 1-5).

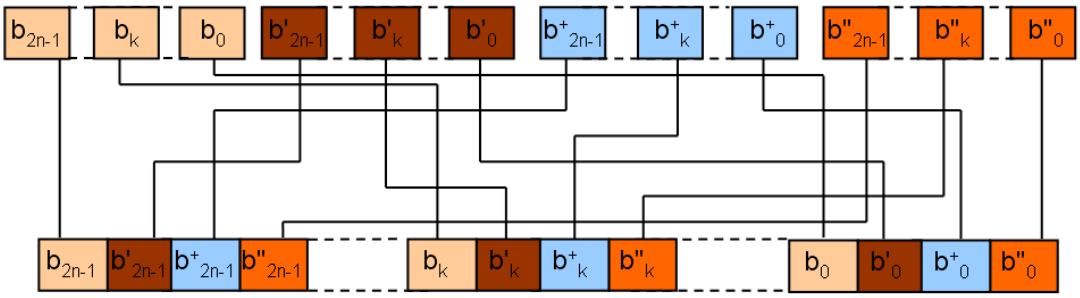

Fig. 1-5 : Schéma d'un multiplexage de colonne avec un facteur 4

Baumann (Texas Instruments) [Bau05] affirme que le facteur de multiplexage doit être de 4 bits au minimum et jusqu'à 8 bits au maximum. Radaelli et al. (Cypress Semiconductor) [RPS+05] pour une mémoire SRAM en technologie 150 nm, et Maiz et al. (Intel) [MHZ+03] pour les technologies SRAM 130 nm et 90 nm, affirment qu'un facteur de multiplexage de 6 est un minimum. Comme il est prévu que le pourcentage et l'étalement des MCU continue d'augmenter, au moins pour les mémoires SRAM, malgré un SER par bit en diminution, il faut s'attendre à devoir employer des facteurs de multiplexage supérieur à 8 pour les technologies les plus récentes afin qu'un MCU se caractérise uniquement par des SBU. Enfin, pour Ibe (Hitachi) [Ibe08], les MBU doivent impérativement être limités à 0-10 FIT par puce.

Le développement de techniques pour protéger les mémoires RAM, en particulier les mémoires SRAM, est donc nécessaire. Ces techniques doivent à la fois être peu coûteuses, mais également tenir compte de l'augmentation des erreurs multiples. Ces erreurs multiples, ainsi que nous l'avons expliqué, étant à la fois plus fréquentes, et impactant des zones de plus en plus importantes.

1.3 Techniques visant à réduire le SER des mémoires RAM

Les techniques dédiées à la réduction des SER pour les mémoires RAM peuvent se regrouper pour l'essentiel en trois catégories : détection instantanée des SEU (capteurs), architectures des mémoires (cellules des mémoires SRAM renforcées), et redondance des données (TMR-ECC). Certaines solutions peuvent être combinées ensemble et/ou complétées par le multiplexage de colonnes.

1.3.1 Détection instantanée des SEU

Les erreurs transitoires peuvent être détectées immédiatement en captant les variations de courant lorsque la valeur d'un bit stocké change à cause d'un SEU. Des capteurs (Built in Current Sensor - BICS) contrôlant les variations de courant dans les colonnes d'une mémoire ont été proposés [CVN95] [GNW+05] [SGB10]. Ces capteurs sont en général couplés avec des bits de parité sur chaque ligne. Les capteurs détectent la colonne du bit fautif, et les bits de parité indiquent sur une ligne si le nombre de bits à 1 à été modifié ou non (le bit de parité est calculé de telle sorte qu'il y ait sur chaque ligne un nombre pair de bits à 1). On connait alors l'emplacement exact de l'erreur. Cette technique peut s'étendre sans problème aux mémoires utilisant le multiplexage de colonnes, en utilisant un bit de parité par mot stocké sur une ligne. Il faut ensuite utiliser un mécanisme particulier pour corriger la donnée à l'intérieur d'une mémoire. En effet il faut recalculer les bits de parité de chaque mot stocké pour détecter le ou les cellules frauduleuses, puis réécrire la donnée en mémoire. Cela peut se faire en parallèle des opérations courantes dans une architecture de type processeur, mais plus difficilement dans une architecture quelconque. De plus, si rien n'interdit d'envisager qu'un capteur proposé dans une technologie donnée soit utilisable dans une autre, actuellement et à notre connaissance, aucune BICS n'a été proposée et/ou validée pour des technologies en dessous des 90 nm.

1.3.2 Architecture des mémoires

Le problème des mémoires SRAM renforcées est le coût à payer (100% de surface en plus en général), ainsi qu'un accroissement de la consommation électrique. De plus certaines de ces mémoires ont eu des performances dégradées par la réduction des dimensions, et ne sont pas forcément résistantes aux MCU, comme celles utilisant la cellule DICE [CNV96]. En effet la cellule DICE est composée de 12 transistors (soit deux fois plus que la SRAM traditionnelle), et a été calculée pour compenser les effets

d'un SEU sur un nœud d'un transistor. Elle peut être employée dans les mémoires SRAM mais aussi pour renforcer les latches. Mais avec la réduction des dimensions, un SEU peut affecter plusieurs nœuds d'une même cellule, ce qui diminue les avantages de cette architecture. Une cellule [SYL11] a été renforcée spécialement pour le *multi node upset* dans une même cellule et ainsi réduire la probabilité de SBU, ce qui logiquement devrait influer sur le taux de MCU. Cette technique est dérivée de la cellule DICE et utilise une cellule mémoire de 13 transistors. Les résultats obtenus par simulation, prédisent des performances excellentes tant que, en cas de *multi node upset*, la charge collectée par un deuxième nœud de la cellule ne devient pas trop importante. Paradoxalement, dans ce dernier cas de figure la cellule DICE possède de meilleures performances. Il existe également une solution dédiée à la prévention des MCU en combinant plusieurs techniques de design de cellule SRAM, mais dont le coût en surface quadruple par rapport à une cellule SRAM à 6 transistors [BD07].

1.3.3 Redondance de donnée

Dans des systèmes critiques tel que l'aérospatial la TMR (Triple Modular Redondancy) appliquée aux mémoires a déjà été utilisée [WJB06]. Les techniques de redondance multiple matérielle, essentiellement double (DMR) ou triple (TMR), consistent à créer plusieurs exemplaires d'une partie d'un circuit (mémoire, registre, fonction logique). Ces modules fonctionnent en parallèle, et les résultats sont comparés. Dans le cas d'une redondance double, une différence détectera une erreur sans la corriger. Dans le cas d'une redondance triple, lorsqu'un résultat diffère, la décision se fait à la majorité (Fig. 1-6). Le surcoût pour le module considéré est donc de 100% pour la DMR, et de 200% pour la TMR.

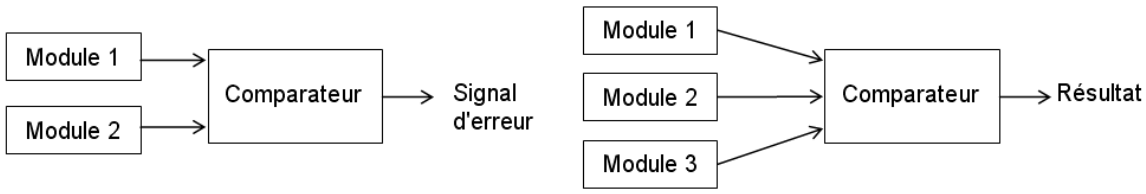

Fig. 1-6 : Schéma général de la DMR (à gauche), et de la TMR (à droite)

La redondance de données peut également se faire sous forme de code correcteur. A *n* bits de données on ajoute *k* bits de contrôle qui permettent ensuite de vérifier l'intégrité des données. La protection contre un seul bit erroné était suffisante dans le passé puisque les erreurs multiples étaient très rares. Par conséquent les codes correcteurs détectant deux erreurs et en corrigeant une seule, comme le code de Hamming étendu (Hamming + bit de parité) et le code de Hsiao [CH84] [Hsi70] étaient très répandus dans les produits industriels [Gra00] [GBT05]. Avec l'apparition des erreurs multiples, l'utilisation du multiplexage de colonnes éloignant le plus possible les bits appartenant à une même donnée, permettait de n'avoir par mot stocké que des erreurs simples [ICS+06]. Mais l'augmentation de la multiplicité des MCU conduit à utiliser des facteurs de multiplexage plus importants, ce qui signifie qu'une même ligne mémoire doit

stocker plus de mots. Mais placer un plus grand nombre de mots sur une même ligne implique une capacité accrue des lignes de la mémoire, une diminution de la vitesse de fonctionnement et une plus grande consommation d'énergie. De plus, le multiplexage de colonnes n'est pas autorisé pour certaines mémoires SRAMS récentes, conçues pour être robustes aux variations des procédés de fabrication actuels, et qui nécessitent de placer cote à cote les bits appartenant à un même mot mémoire [CMN+08] [SVC09].

[BBN+07] a montré que les codes correcteurs double et Triple (de type BCH binaire) et qui ont un surcoût en terme de surface inférieure à 100% par rapport à une mémoire non protégée sont plus efficaces en terme de probabilité d'erreurs, à la fois que le code de Hamming et que la TMR. Ils sont de plus moins coûteux en surface que la TMR, dont le surcoût est de 200%. IBM dans la mémoire cache de son processeur z990 [MSS+07] utilise un code correcteur corrigeant un symbole (groupe de bits) composé de deux bits, et Intel dans le processeur Itanium utilise un code double correcteur et triple détecteur [RBB+11].

1.4 La protection des fonctions logiques

Comme nous l'avons déjà dit, quand le nœud appartient à une porte logique, l'impulsion électrique est transformée en une impulsion en tension (SET). Cette impulsion parasite peut induire un SEU si elle est capturée par un composant de stockage. Leur fréquence d'apparition est plus difficile à mesurer car elle dépend de l'architecture, et de l'état du circuit au moment où apparait un SEU. Toutefois leur fréquence d'apparition augmente, ce qui conduit, tout comme les mémoires, à concevoir des systèmes de protection dédiés aux fonctions logiques.

1.4.1 Propagation d'une erreur dans une fonction logique

Une impulsion parasite peut être propagée par un ou plusieurs chemins de portes logiques reliées en série pour atteindre un latch ou une flip-flop. La propagation de l'impulsion est conditionnée par l'état de la logique combinatoire, puisqu'une valeur de contrôle (un 0 pour une porte NAND ou un 1 pour une porte NOR) sur l'entrée d'une porte bloquera la propagation d'une erreur à ce niveau (Fig. 1-7). Ainsi, une impulsion parasite peut être propagée seulement par un chemin sensibilisé, c'est-à-dire, à travers un chemin tel que les entrées non concernées par l'impulsion parasite ne masquent pas la propagation de celle-ci.

La propagation d'une impulsion est également conditionnée par sa durée. Avec une bonne approximation nous pouvons dire que si la durée d'une impulsion parasite est inférieure à la période de transition d'une porte logique, elle ne peut être propagée au travers de celle-ci. Si cette durée excède le double de la période de transition d'une porte logique, l'impulsion est propagée sans atténuation. L'impulsion est propagée avec une certaine atténuation si sa durée est comprise entre ces deux valeurs [Nic05]. Enfin, quand l'impulsion atteint un latch ou une bascule, elle sera capturée pour produire une erreur transitoire seulement si elle coïncide avec une transition d'horloge. Ces conditions

rendent les erreurs transitoires dues aux SET plus rares que les SEU induits par des particules heurtant les nœuds d'une cellule de stockage.

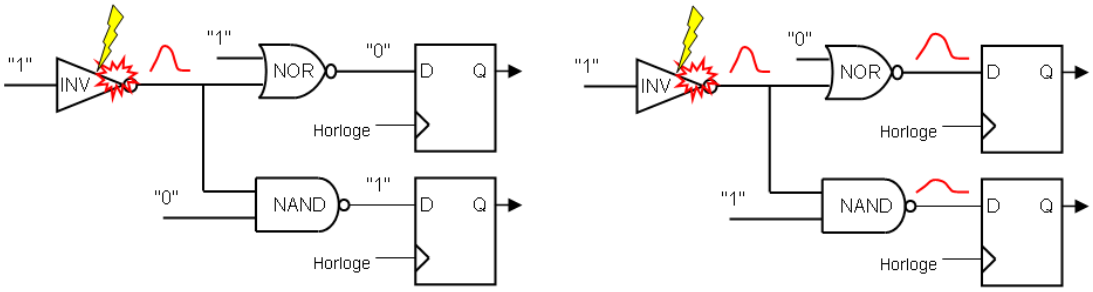

Fig. 1-7 : Masquage d'une erreur transitoire (à gauche), propagation d'une erreur transitoire (à droite)

Cependant, la fréquence de ce type d'erreur augmente aussi pour plusieurs raisons. Le temps de transition des portes logiques devenant extrêmement court, les durées des impulsions parasites deviennent plus grandes et ne sont plus filtrées par les caractéristiques électriques des portes. En outre, comme les fréquences deviennent très élevées, la probabilité que des impulsions parasites coïncident avec une transition d'horloge devient également importante [BB97] [BBB+97] [BEA+04]. Les impulsions parasites peuvent avoir, avec une probabilité assez grande, une durée de 500 ps à 900 ps pour un circuit combinatoire dans une technologie 90 nm. Et il est prévu que cette durée continue d'augmenter avec la réduction des dimensions [NBS+07]. Une explication de ce phénomène peut venir de la réduction de l'alimentation des portes logiques : l'énergie nécessaire pour créer une impulsion parasite diminue.

Pour finir, l'état de sensibilisation des chemins peut ne pas avoir un impact significatif sur la réduction de la sensibilité d'un circuit aux SEU. Un nœud d'un circuit combinatoire est habituellement relié à des bascules par plusieurs chemins. Ainsi, la probabilité qu'aucun d'eux ne soit sensibilisé peut être importante. En effet, l'impulsion sera souvent propagée par plusieurs chemins, résultant en plusieurs impulsions qui auront affecté le circuit à différents moments. Ceci augmente la probabilité qu'une partie de ces impulsions coïncide avec une transition d'horloge lorsqu'elle atteint une bascule. Des techniques de type BIST existent pour protéger les portes logiques qui détectent les variations de charges causées par le choc des particules. Elles conduisent à une augmentation de délai de 4%, et de surface de 100% par porte protégée [GJK+06]. En faisant un trie en fonction de leur importance dans un circuit des portes à protéger, la surface en plus est de 30 à 50% en moyenne sur l'ensemble d'un circuit. Ce surcoût varie en fonction du dit circuit entre 8% et 130%. D'autres capteurs de courants, connectés au niveau du substrat des transistors qui composent les portes, et qui peuvent être commun à plusieurs dizaines de transistors, ont permis de réduire le surcoût en surface. Ces capteurs, suivant les paramètres d'implémentation choisie, conduisent à une surface supplémentaire variant entre 2% et 50%, et peuvent détecter des impulsions parasites d'une durée de quelques picosecondes [LKN+07].

1.4.2 Détection des erreurs par redondance temporelle

Ces impulsions se traduisent en fait par une erreur assimilable à une faute de délai, puisque le circuit revient dans un état stable après coup. D'un certain point de vue elle est comparable aux erreurs de délai dues aux variations de tensions, de températures et au vieillissement accéléré des composants, qui font que le circuit ne peut plus temporairement ou définitivement fonctionner à la même fréquence. Pour cette raison, il existe un nombre conséquent de solutions visant à détecter et/ou corriger les fautes de délais (ou les erreurs transitoires). Citons, et cette liste n'est pas exhaustive, l'architecture RAZOR [END+03] (Fig. 1-8), l'architecture GRAAL [Nic07] (Fig. 1-9), ainsi que [HLS08] [ASS09] [FNM09] [ZMM+06] et [BKL+08]. Ces solutions utilisent souvent des bascules D (ou latches) supplémentaires, enregistrant une donnée à plusieurs instants différents. De plus le signal d'erreur généré est souvent utilisé pour rectifier l'erreur en choisissant la donnée stockée par la bascule additionnelle, plutôt que celle stockée par la bascule originale.

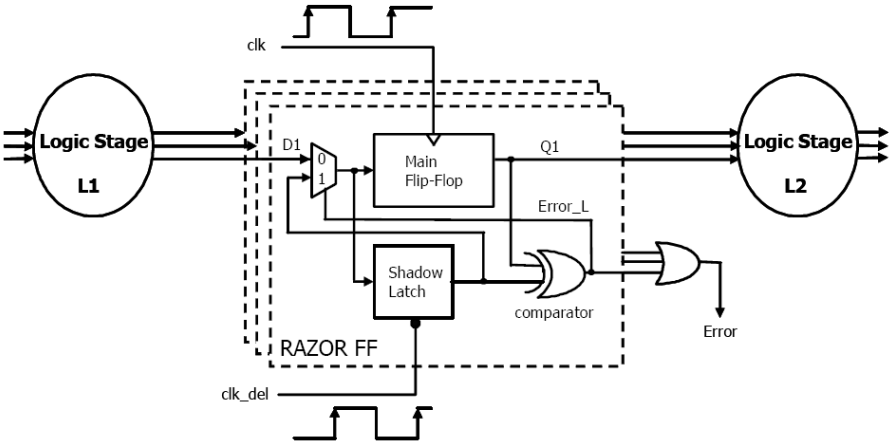

Fig. 1-8 : Architecture de type RAZOR [END+03]

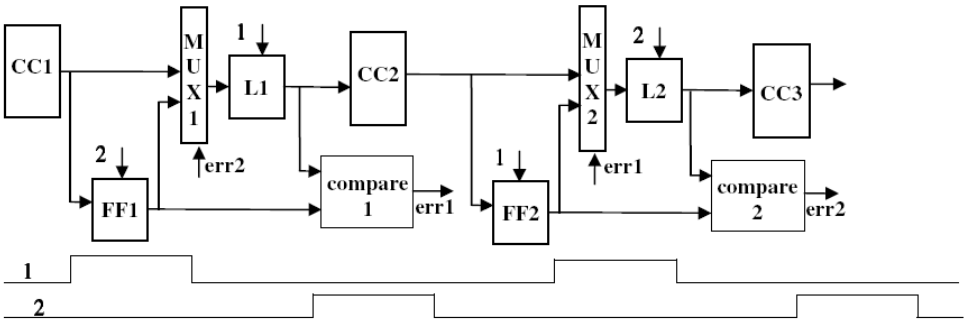

Fig. 1-9 : Architecture de type GRAAL [Nic07]

La majorité de ces techniques a toutefois l'inconvénient de ne fonctionner que pour des erreurs transitoires dont les durées correspondent à une fraction de la période d'horloge. C'est pourquoi elles sont surtout utilisées pour la détection des fautes de délais, plutôt que pour la détection des impulsions parasites. En effet elles utilisent un système de détection basé sur deux horloges ou plus décalées dans le temps. Ainsi on peut constater, sur une fraction de période d'horloge, si l'état en entrée d'un composant de

stockage a changé en stockant le résultat d'une fonction logique à T et T+δ. Pour les architectures, telles que les processeurs, fonctionnant à des fréquences au-delà du gigaHertz, et compte tenu de la durée possible des erreurs transitoires, cela limite les capacités de détection. Dans les processeurs, peuvent alors être employées d'autres techniques basées sur une répétition des opérations effectuées consécutivement (ou parallèlement dans le cas des multiprocesseurs) [YK11].

1.4.3 Détection des erreurs par redondance matérielle

Plutôt que d'effectuer une détection/correction, basée sur une détection temporelle, il est possible, de la même manière que pour les mémoires, d'utiliser le principe de la redondance matérielle (DMR ou TMR, Fig. 1-6). L'inconvénient étant de façon immédiate une augmentation de la surface totale de 100 ou 200% du circuit combinatoire. De manière analogue aux ECC utilisés dans la protection des mémoires, des systèmes de codage des données, fonctions logiques approximées et prédiction des résultats existent. Les premiers circuits testant l'intégrité des fonctions logiques étaient basés sur des codes arithmétiques [Pet58] [PW72], la prédiction du bit de parité [SHB68] [GR68], puis les codes de Berger [LTR+92]. Le schéma général d'un circuit doté d'un système de détection incorporé est donné par la figure 1-10. Il faut noter que la présence du bloc "circuit approximé" dépend du type de solution utilisée.

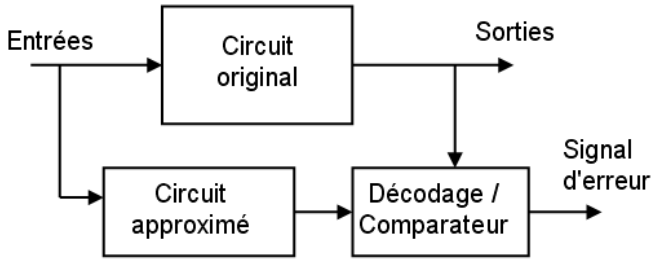

Fig. 1-10 : Circuit logique avec un système de détection incorporé

Toutefois, les circuits de codage et de détection complexes augmentent la surface de la logique combinatoire, et certaines erreurs ne sont pas détectables [LT70]. Si l'erreur se produit en entrée du circuit par exemple, l'erreur sera propagée à travers l'ensemble circuit original plus le système de détection. Ce dernier ne verra alors qu'une opération effectuée par une autre donnée que celle qui aurait du être utilisée, et le résultat sera compté comme correct. L'utilisation de ces techniques peut augmenter la surface de 35 à 171% [TM94]. Ces techniques, optimisées pour les opérations arithmétiques, donnent des résultats où la surface est augmentée de 15 à 24% pour les additionneurs, et de 40 à 50 % pour les multiplieurs [NDM+97] [ND99] [DNB+98] [Nic03] [OGS+03] [OMS+04]. Plus récemment des solutions avec un taux de couverture d'erreurs évalué entre 80 et 95%, et qui nécessitent seulement un surcoût de 25% en surface et de 35% de consommation ont été proposées [CM08] [VJA+08].

1.4.4 Fiabilité des bascules

La fiabilité des bascules de type latch et flip-flop est devenue comparable à celle des cellules des mémoires SRAM de par leurs architectures similaires (Fig 1-11). Toutefois leur taux de vulnérabilité, tout comme les fonctions logiques sont difficiles à déterminer avec précision, car il dépend de la fréquence et de l'architecture. A cause des similitudes existant avec une cellule mémoire SRAM, les solutions pour les fiabiliser sont assez proches de celles employées pour les SRAM (BIST, TMR, cellule renforcée de type DICE) déjà citées. A noter que le coût de l'utilisation de ces techniques est d'autant moins important que les composants de type latch/flip-flop représentent une part du circuit en général beaucoup moins importante que les mémoires RAM. Ainsi la technique TMR dont le surcoût est de 200% est très coûteuse lorsqu'elle est appliquée à des mémoires SRAM qui peuvent représenter plus de 70% de la surface totale. En revanche le surcoût de la TMR appliquée au latch/flip-flop est à pondérer par la fraction de surface du circuit occupée par ces composants.

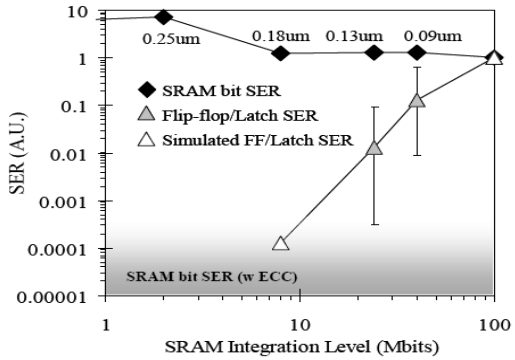

Fig. 1-11 : Evolution du SER par bit des SRAM et des bascules en fonction de la technologie [Bau05]

1.5 Conclusion

La fiabilité des circuits est un des problèmes majeurs de la micro-électronique aujourd'hui. Ce problème concerne aussi bien les composants de stockage massif que la partie opérative des circuits. Les mémoires RAM sont les premières concernées par ces phénomènes. Bien que leurs sensibilités par cellule ait tendance à diminuer, l'augmentation de leur densité maintien leur taux de SEU dans le cas des mémoires DRAM, et l'augmente dans le cas des mémoires SRAM. De plus cela conduit à devoir prendre en compte des erreurs multiples, avec un nombre de bit faux par évènement qui augmente de manière importante. Les simulations prédisent qu'une erreur sur deux sera multiples, et peuvent affecter plus d'une dizaine de bits sur une même ligne. De fait la protection de ces mémoires à moindre coût, et sans dégrader les performances du système constitue la contribution principale de ce mémoire. La seconde consiste à partir d'un système comportant un système de détection d'erreur intégré (capteur, redondance matérielle ou temporelle), de ré-exécuter les derniers cycles du circuit. L'erreur étant transitoire, elle ne sera pas répétée.

Chapitre 2. Architecture matérielle pour supprimer les délais dus à l'utilisation d'un code correcteur pour protéger une mémoire

---

L'utilisation d'un code correcteur pour protéger une mémoire embarquée peut provoquer une perte de performance inacceptable dans bon nombre d'application en terme de débit des données. Ce chapitre présente une méthode architecturale universelle pour résoudre ce problème. Cette solution élimine le délai dû à l'utilisation d'un code correcteur lors de l'écriture d'une donnée en mémoire en utilisant deux mémoires séparées. La pénalité de temps est limitée au seul cas où une erreur est effectivement détectée lors de la lecture. Cette technique effectue la décontamination du circuit après qu'une donnée corrompue ait été lue. La contrepartie est de devoir ajouter des FIFO supplémentaires pour restaurer le contexte du circuit au moment où la donnée erronée a été lue pour effectuer une décontamination correcte. Cette technique s'insère aisément pour une mémoire connectée à une architecture régulière, et nécessite quelques ajustements pour être utilisée dans les architecture les plus complexes. Elle a été testée dans une architecture de modulateur OFDM avancé. Les simulations temporelles ont démontré son efficacité, avec un coût matériel et énergétique relativement faible.

2.1 Rappel sur les codes correcteurs

Les codes correcteurs d'erreurs font partie des techniques de protection des mémoires les plus couramment utilisées. Ces codes peuvent être employés dans la protection des données qui sont stockées dans des mémoires, comme celles transmises dans les communications numériques. Ces codes sont basés sur la redondance d'information : l'information stockée ou transmise est répétée, souvent de façon plus compacte (donc incomplète). Il est dès lors possible de détecter une altération de l'information (code détecteur d'erreurs), et d'effectuer la correction (code correcteur d'erreurs). On peut distinguer dans l'ensemble des codes numériques, les codes linéaires et les codes non linéaires. Dans le cadre de la protection des mémoires, les codes correcteurs employés sont en général, des codes linéaires et de dimension finie. Ces codes ont une complexité variable en fonction du nombre d'erreur qu'ils peuvent détecter ou corriger.

Un code linéaire de dimension finie peut se caractériser par une fonction de codage *g* qui transforme un mot *x* de *n* bits en un mot y de *n+k* bits.

$$g : \{0, 1\}^n \rightarrow \{0, 1\}^{n+k}$$

$$x \quad \rightarrow y = g(x)$$

La fonction g définit pour un élément de $\{0, 1\}^n$ une image unique dans l'ensemble $\{0, 1\}^{n+k}$. Ce sont les "*mots du code*" qui ont un unique antécédent dans $\{0, 1\}^n$. Certains éléments de $\{0, 1\}^{n+k}$ n'ont pas d'antécédent dans $\{0, 1\}^n$: ils sont considérés comme des mots faux. C'est-à-dire qu'un mot qui ne fait pas partie de $g(\{0, 1\}^n)$ sera interprété comme étant un "mot du code" comportant des erreurs. Pour la correction, une fonction f permet la correction des données avec :

- $f(y) = g^{-1}(y)$ si y est un mot du code ;
- pour un élément *y* de $\{0, 1\}^{n+k}$ qui ne fait pas partie du code, on cherche un élément *y'* de $\{0, 1\}^{n+k}$ qui minimise la *distance de Hamming* avec *y* (c'est-à-dire qui minimise le nombre de différences avec *y*, autrement dit le nombre d'erreurs). Si cet élément *y'* est unique, la ou les erreurs sont corrigeables et lors du décodage on choisira $f(y) = g^{-1}(y')$. Si cet élément *y'* n'est pas unique, les erreurs ne sont pas corrigeables puisqu'on ne peut pas retrouver le mot d'origine.

Le code détecteur d'erreurs le plus simple, qui permet de détecter une seule erreur ou plus généralement de faire la différence entre un nombre pair et impaire d'erreurs est le bit de parité. Le principe est de compter le nombre de bits à 1. S'il y a un nombre pair de bits ayant pour valeur 1 on rajoute un bit à 0 ; s'il y a un nombre de bits impair on rajoute un 1. Au final, en incluant le bit de parité, tous les mots stockés ont un nombre pair de bits à 1 stockés en mémoire. Le code correcteur le plus simple, permettant de corriger une seule erreur avec le minimum de bits supplémentaires, est celui de

Hamming. Ce code a toutefois l'inconvénient de créer parfois une confusion entre une erreur simple et une erreur double. Lorsqu'une erreur double est interprétée comme une erreur simple, cela aggrave le problème en rajoutant une troisième erreur après correction d'un bit juste interprété comme faux. Pour éviter cela on rajoute un bit de parité qui permet de faire la distinction entre les deux cas. En effet, le bit de parité indiquera si un nombre pair (incluant zéro) ou impair de bits ont été modifiés. Le code de Hsiao possède les mêmes capacités de détection et de correction que le code de Hamming avec bit de parité, mais avec de meilleures performances en termes de coût de surface et de consommation (pour les fonctions de codage et de décodage) [CH84] [Hsi70]. Toujours avec le même nombre de bits que le code de Hsiao, et en supposant que, en cas d'erreur double, les deux bits fautifs sont placés cote à cote, il existe des codes corrigeant les erreurs simples ainsi que les erreurs doubles adjacentes. Il existe toutefois un risque de confusion dans le cas d'erreur double non adjacente [DT07]. En dehors du coût variable suivant le code utilisé (Hamming+bit de parité, Hsiao ou [DT07]) des fonctions de codage et de décodage, le coût supplémentaire (la quantité de mémoire nécessaire pour stocker les données codées) est de 37,5% en plus pour 16 bits de données à coder, 22% pour 32 bits, et 12,5% pour 64 bits.

Des familles de codes linéaires permettent de choisir à l'avance les capacités de détection et de correction du code employé. C'est le cas des codes BCH (Bose-Chaudhuri-Hocquenghem) qui font partie de la famille des *codes cycliques*. De la même manière, la capacité de stockage supplémentaire nécessaire diminue en proportion avec l'augmentation du nombre de bits à coder. Pour un code BCH binaire double correcteur, le surcoût en mémoire est de 62,5% pour 16 bits, de 37,5% pour 32 bits, et 22% pour 64 bits. Pour un code BCH binaire triple correcteur, le surcoût est de 94% pour 16 bits, 56% pour 32 bits, et 33% pour 64 bits. Il existe également des codes BCH non binaires, plus complexes, et donc plus coûteux en surface en consommation, mais aux capacités de corrections plus importantes, comme les codes Reed-Solomon.

Quand les mémoires ne devaient être protégées que contre des SBU, les codes correcteurs d'erreurs simples (corrigeant une seule erreur) suffisaient. Ces codes ont été largement employés dans les produits industriels [Gra00] [GBT05]. Avec l'apparition des risques de MBU, le multiplexage de colonnes a permis de ne garder que des erreurs simples. Cette technique ne peut toutefois pas être utilisée dans certaines mémoires récentes dessinées pour être robustes aux variations de processus de fabrication (comme mentionné au chapitre 1). De plus même dans les mémoires permettant d'utiliser cette technique, cela requiert une plus grande capacité de stockage par ligne, une augmentation de la consommation électrique, et une diminution de la vitesse de fonctionnement. Ces inconvénients s'aggravent avec l'augmentation du facteur de multiplexage nécessaire pour éviter les MBU.

2.2 La pénalité de temps dû à l'utilisation de codes correcteurs

L'un des problèmes posé par l'utilisation des codes correcteurs est le délai important ajouté par le circuit de codage avant écriture des données en mémoire, et par les circuits de détection et correction d'erreurs après la lecture des données en mémoire (Fig.2-1). Ces délais sont susceptibles d'induire une importante réduction de la fréquence d'horloge qui peut être inacceptable dans un certain nombre d'applications. En général, plus un code est complexe plus celui-ci augmente les délais d'écriture et de lecture en mémoire. Il peut être nécessaire dans un circuit d'éliminer ces délais supplémentaires pour l'une ou les deux opérations. La solution classique, consistant à effectuer le codage et le décodage des données en plusieurs cycles d'horloge n'est pas toujours acceptable.

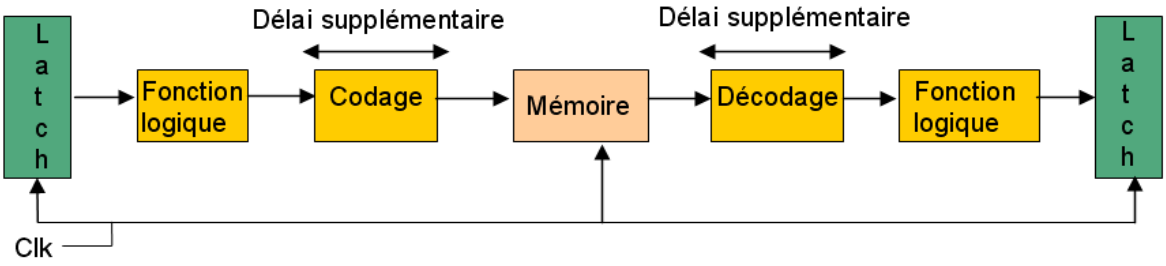

Fig. 2-1 : Illustration de la pénalité de temps due au code correcteur

La solution classique pour ne pas diminuer la fréquence de fonctionnement du circuit consiste à rajouter des étages de pipeline dans les circuits de codage et de décodage. Mais cela implique une augmentation de la latence, puisqu'un ou plusieurs cycles d'horloge supplémentaires sont nécessaires pour effectuer ces opérations. Dans les systèmes à base de processeurs ce n'est en général pas un problème majeur, puisqu'il est possible d'utiliser immédiatement la donnée, sans contrôler son intégrité, et de ré-exécuter l'instruction en cas d'une détection d'erreur [SP88] [SKF01] [MB05] [CRS+05]. Dans les architectures n'utilisant pas de processeurs, il n'existe pas de techniques génériques, permettant de limiter l'impact en termes de latence sur la fréquence d'horloge, ou la lecture des données. Suivant les algorithmes traités, dans le cas où les données doivent faire plusieurs fois des aller retour entre la fonction logique et la mémoire, le coût peut être important, voir inacceptable dans des applications où le facteur temps est critique.

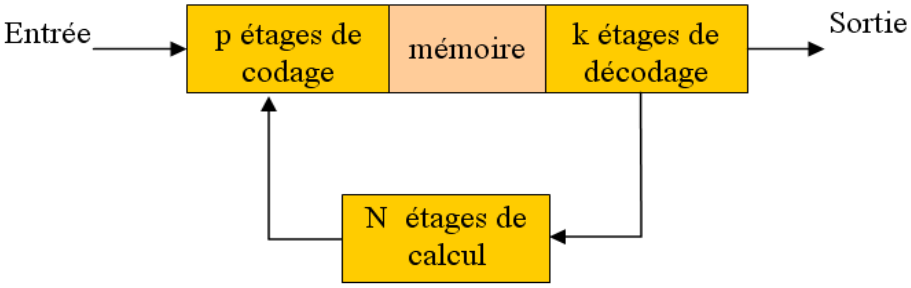

Fig. 2-2 : Réduction du débit dans un circuit utilisant une mémoire protégée par un code correcteur

Imaginons un circuit traitant des données pendant n cycles, une mémoire utilisant des codes correcteurs avec p cycles de codage et k cycles de décodage (Fig. 2-2). Le nombre de cycles perdus est fonction du nombre de boucles effectuées dans l'architecture (sans prendre en compte les cycles pendant lesquels les données sont stockées) :

- 0 boucle : 0 cycle de calcul + $p+k$ cycles perdus ;
- 1 boucle : N cycles de calcul + $2\times(p+k)$ cycles perdus ;
- M boucles : $M\times N$ cycles de calcul + $(M+1)\times(p+k)$ cycles dus au passage des données en mémoire.

Ici nous avons représenté un système sous forme de boucle. Le problème est identique si on considère M circuits de calculs successifs, chacun utilisant un buffer protégé par un code correcteur. Ces buffers stockent les résultats des opérations intermédiaires qui seront utilisés par le circuit suivant. Le seul moyen de réduire cet inconvénient, sans modifier l'architecture du circuit, est de réduire le nombre de cycles d'horloge pendant lesquels chaque donnée est stockée en mémoire de $p+k$ cycles d'horloge. Toutefois, suivant l'algorithme implémenté et les valeurs des nombres p et k, cela n'est pas toujours possible. En particulier, en cas d'éventuels problèmes de dépendance de données, cette perte peut être plus importante.

2.3 Séparation des bits de données et des bits de contrôle

Une des solutions pour réduire l'importance des délais dus aux codes correcteurs est de faire subir un traitement différent aux bits codés. En effet, en règle générale, les codes correcteurs utilisés dans le cadre de la protection des données stockées en mémoire sont linéaires et de dimensions finies. Cela permet de modifier la manière de coder une donnée, tout en conservant les même capacités de détection/correction.

Le code étant linéaire et de dimension finie, la fonction g de codage d'un mot x, de n bits, en un mot y, de $n+k$ bits, peut alors s'écrire sous forme matricielle :

$$Y = G.X$$

G est la matrice de codage de dimension $(n+k, n)$ caractéristique du code correcteur utilisé ($n+k$ lignes et n colonnes). X (respectivement Y) est le vecteur de n bits (respectivement $n+k$ bits) correspondant au mot x (respectivement y). La matrice G peut alors se mettre sous la forme d'une matrice équivalente G', où G' est de la forme :

$$G' = \begin{pmatrix} I_n \\ R \end{pmatrix}$$

I_n étant la matrice identité de taille n, et R la sous-matrice de dimension (k, n) caractéristique du code. On dit que la matrice G' est écrite sous forme *systématique*. Le codage d'une donnée s'écrit alors :

$$G'.X = \begin{pmatrix} X \\ R.X \end{pmatrix}$$

$G'.X$ peut donc se diviser en n bits de données X, et k bits de contrôle $R.X$. De la même manière l'opération contrôlant l'intégrité des données peut s'écrire, $S=H.Y$, où H est la matrice de contrôle de taille $(k, n+k)$, et S le syndrome de l'erreur. En cas d'absence d'erreur ou si l'erreur est non détectable $S=0_k$. Sous forme *systématique*, H s'écrit :

$$H=(R\ I_k)$$

En cas d'absence d'erreur on a bien $S=(R\ I_k).^t(^tX\ ^tRX)=RX+RX=0$. Rappelons ici que du point de vue de la théorie des codes correcteurs, l'opération + est équivalente au *XOR* en électronique. Dans le cas d'un syndrome non nul, ce dernier est traité par une fonction qui :

1– indique si l'erreur est corrigeable (nombre d'erreurs a priori inférieur ou égal à la capacité de correction du code)

2– dans le cas où l'erreur est corrigeable, cette fonction calcule un vecteur correcteur V_C pour effectuer la correction du vecteur X' défectueux $X=X'+V_C$.

A retenir :

- Un code correcteur mis sous forme systématique peut se caractériser par une matrice R aussi bien lors du codage que lors du contrôle de l'intégrité des données.
- Les bits de données X et les bits de contrôle RX sont séparables.
- Pour contrôler l'intégrité des bits de données lors de la lecture, il suffit de recalculer les bits de contrôle RX et de les comparer aux bits de contrôle précédemment calculés au moment du codage des données.

A ce stade nous pouvons déjà dire qu'il n'est pas nécessaire de stocker les bits de données et les bits de contrôle dans la même mémoire. Ils peuvent être stockés dans deux mémoires séparées, ce qui conduit à l'architecture matérielle suivante (Fig. 2-3). Les bits de données et de contrôle sont stockés dans une mémoire data et une mémoire code. Nous pouvons voir dans cette figure le bloc *Codage* calculant les bits de contrôle à partir des bits de données qui doivent être enregistrés en mémoire, ainsi que le bloc *Codage* calculant les bits de contrôle à partir des données lues en mémoire. Ces derniers bits de contrôle sont comparés avec les bits de contrôle stockés dans la mémoire code par un banc de porte *XOR (XOR1)*, qui produit le syndrome de l'éventuelle erreur. Ce syndrome indique la position du bit faux en code binaire. Ce syndrome est transformé en un mot

ayant le même nombre de bits que les données et les bits de contrôle lus en mémoire, les bits à 1 correspondant à l'emplacement des erreurs. Ensuite un second banc de porte *XOR (XOR2)* procède à la correction éventuelle des erreurs. Cela ne modifie, dans le mot composé des bits de données et des bits de contrôle, que les bits situés au même emplacement que les bits à 1 dans le mot issu du syndrome.

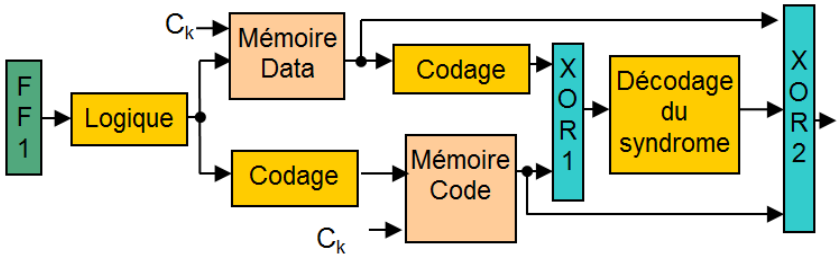

Fig. 2-3 : Implémentation des codes correcteurs utilisant des mémoires séparées

Ce principe a été proposé dans [Nic05], et exploité dans [MB05] où la détection et la correction des données se font de façon indépendante de leur lecture (Fig. 2-4). Cette architecture à base de processeurs, utilise un microcontrôleur (FRAC) pour faire le lien entre la mémoire et le processeur principal. Ce microcontrôleur gère les interruptions en cas d'erreur, et la réémission des données nécessaires à la réexécutions des instructions interrompues. Dans le cas où la mémoire n'est pas utilisée, ce système possède en plus un mode de scan périodique vérifiant et corrigeant l'intégrité des données en mémoires.

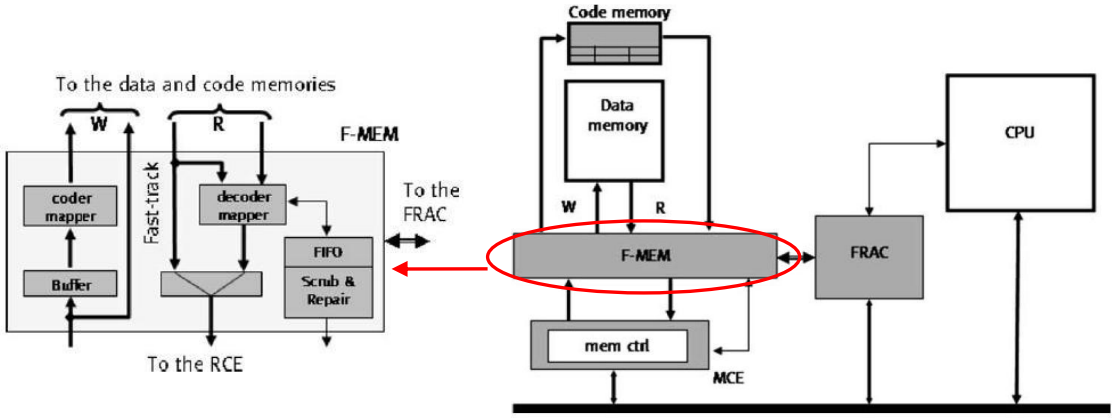

Fig. 2-4 : Architecture présentée dans [MB05]

En dehors de la latence du microcontrôleur, cette technique n'implique pas de délai supplémentaire pour l'écriture, ou la lecture des données en mémoire. Toutefois cette latence subsiste, mais est négligeable du fait qu'il s'agit d'une architecture de type processeur. À notre connaissance, il n'existe pas de technique destinée à limiter la latence des données lors de l'écriture, et de la lecture en mémoire, dans toutes les architectures autres que celles de type processeur. La suite de ce chapitre présente la technique que nous proposons. Elle permet de résoudre efficacement ce problème, en partant du schéma général de la figure 2-3.

2.4 Suppression du délai lors de l'écriture des données en mémoire

Pour supprimer le délai dû au code correcteur utilisé lors de l'écriture des données en mémoire, on va se servir de la propriété des codes correcteurs linéaires mis sous forme systématique présentée précédemment. En effet lors du codage des données, les bits de données et les bits de contrôle peuvent être séparés. Il y a ensuite plusieurs solutions pour éliminer la pénalité de temps dû au calcul des bits de contrôle, comme expliqué ci-après.

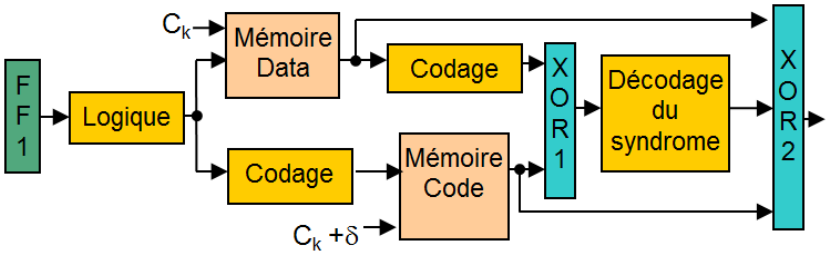

Fig. 2-5 : Architecture utilisant des mémoires séparées, avec signal d'horloge décalée

La figure 2-5 représente le schéma de notre solution pour éliminer le délai supplémentaire dans la partie écriture. La différence entre la figure 2-5 et la figure 2-3 est due à l'utilisation d'une horloge décalée pour la mémoire code. Cette deuxième horloge est décalé d'une fraction de cycle d'horloge ($0 \leq \delta < 1$). Dans cette architecture, les bits de données sont écrits en mémoire dès qu'ils sont disponibles. Ainsi la fréquence de l'horloge (signal C_k) est la même que dans le cas d'un système n'utilisant pas de code correcteur. De cette façon la pénalité de temps lors de l'écriture en mémoire est éliminée. Il ne subsiste que le délai du calcul des bits de contrôle. Pour résoudre ce problème, plusieurs solutions sont possibles.

Solution 1 : La mémoire code utilise un signal de même fréquence que le signal C_k, mais décalé dans le temps $C_k+\delta$, et le délai δ est supérieur ou égal au délai dû au calcul des bits de contrôle. Ainsi les bits de contrôle sont écrits en mémoire dès qu'ils sont disponibles. Le signal d'horloge décalé $C_k+\delta$ peut être généré localement, en ajoutant un composant de retard sur le chemin du signal arrivant à la mémoire code par rapport au signal arrivant à la mémoire de données. Une implémentation minutieuse de cet élément de délai considèrera, à la fois le pire cas du bloc calculant les bits de contrôle, ainsi que le meilleur cas du composant de retard. Pour cette solution $0 < \delta < C_k$.

Solution 2 : Une autre possibilité est d'utiliser le même signal pour les deux mémoires. L'une des mémoires utilisera le front montant du signal d'horloge, et l'autre utilisera le front descendant. Cela fonctionnera correctement tant que le délai du bloc de calcul des bits de contrôle n'excède pas l'intervalle entre ces deux fronts d'horloge. En général, dans ce cas de figure, $\delta = 0{,}5\ C_k$.

Solution 3 : Une autre solution pour implémenter cette architecture, très utile lorsque le délai du au calcul des bits de contrôle est très grand, consiste à rajouter des étages de pipeline dans le bloc de calcul. Avec cette solution l'enregistrement des bits de contrôle se fera un ou plusieurs cycles d'horloge plus tard par rapport à l'enregistrement des bits de données. Ce nombre de cycles supplémentaires correspond au nombre d'étages de pipeline ajouté pour répartir le délai de calcul sur plusieurs cycles d'horloge sans diminuer la fréquence. Les deux mémoires utilisent par contre le même signal d'horloge ($\delta=0$). Toutefois, l'enregistrement des bits de contrôle se faisant avec un ou plusieurs cycles d'horloge de retard, l'adresse de stockage en mémoire doit être conservée durant le même nombre de cycle.

Dans le cas où le signal d'horloge retardé est utilisé à la fois pour la lecture et l'écriture en mémoire, les bits de contrôle seront lus avec un délai par rapport aux données lues en mémoire. Mais ce délai n'augmente pas celui dû à la détection et à la correction d'une erreur. En effet, les données en mémoire doivent d'abord passer dans un autre circuit servant à recalculer les bits de contrôle. Ils seront ensuite comparés à ceux lus dans la mémoire code alors que ces derniers sont directement prêts pour cette opération. Le délai pour recalculer les bits de contrôle étant le même que durant la phase d'écriture (δ), les bits de contrôle et les bits de données sont synchrones au moment de leur comparaison. Le raisonnement est le même dans le cas où l'une des mémoires utilise le front d'horloge montant et l'autre le front d'horloge descendant. Dans le cas où un certain nombre d'étages de pipeline a été ajouté pour que le bloc de codage ne réduise pas la fréquence de l'architecture, le même nombre d'étages sera ajouté pour le bloc qui doit recalculer les bits de contrôle pour le calcul du syndrome.

A retenir : Dans tous les cas de figure, le délai au niveau de la mémoire sauvegardant les bits de contrôle est donc compensé au moment de la lecture, à cause du nouveau calcul de ces derniers. Cette technique n'aggrave donc pas le délai lors de la lecture des données en mémoire. L'objectif est donc atteint, à savoir que les bits de données sont stockés sans délai supplémentaire dans la mémoire, malgré l'utilisation d'un code correcteur. Le délai le plus important, et aussi le plus difficile à supprimer reste toutefois celui existant lors de la lecture des données en mémoire.

2.5 Elimination du délai lors de la lecture des données en mémoire

Le délai supplémentaire lors de la lecture est dû au nouveau calcul des bits de contrôle pour obtenir le syndrome, et à la correction d'une erreur éventuelle. De la même manière que pour les opérations d'écriture en mémoire, on propose ici d'effectuer le contrôle de l'intégrité des données en parallèle de l'utilisation des données lues par le reste du circuit. Toutefois, l'implémentation de la solution dépend de divers cas de figure, et en particulier de l'architecture du circuit. Cela rend l'élimination du délai lors de la lecture d'une donnée plus complexe que lors de l'écriture d'une donnée en mémoire. Dans

la suite de cette section nous allons détailler les diverses implémentations de cette solution pour une architecture simple. Les cas plus complexes seront abordés dans la section suivante.

La figure 2-6 représente le schéma d'un système où les données lues dans la mémoire passent dans un circuit logique (*Logique 1*) et sont stockées dans un banc de flip-flops (*FF1*). Ce bloc logique peut très bien être vide (i.e. les données sont directement stockées dans *FF1*). Il peut également recevoir des données provenant d'un autre endroit du circuit (*Entrée 1*). Le reste du circuit n'est qu'une succession plus ou moins longue de fonctions logiques et d'étages de pipeline correspondant à ce même schéma. Pour l'implémentation de notre solution, ce type d'architecture sera qualifié de *régulier*. Dans cette figure la mémoire n'est pas protégée par un code correcteur. Si un code correcteur est utilisé, le circuit de détection et de correction sera placé entre la mémoire et le circuit combinatoire (*Logique 1*), augmentant ainsi les délais et diminuant potentiellement la fréquence de fonctionnement. Pour résoudre ce problème, on peut ajouter un ou plusieurs étages de pipeline (i.e. ajouter un ou plusieurs étages de bascules entre la mémoire et *FF1* dans la figure 2-6). Cette solution permettra au circuit de fonctionner à la fréquence désirée. Cependant les performances du système seront quand même diminuées puisque les données lues mettront un ou plusieurs cycles d'horloge supplémentaires pour atteindre *FF1*, ceci par comparaison avec le circuit non protégé. Cela peut constituer un problème dans beaucoup d'applications, ou parties d'applications, dans lesquelles le facteur temps est critique.

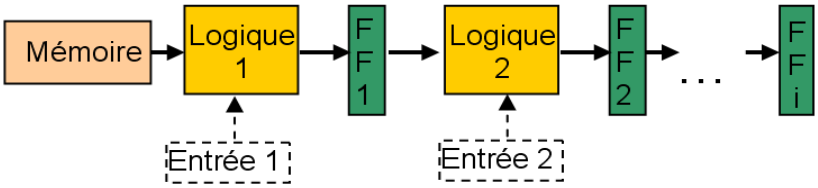

Fig. 2-6 : Mémoire non protégée connectée à un pipeline

Pour supprimer la pénalité de temps, la solution consiste à court-circuiter les circuits de détection et de correction. Il n'y a alors plus de délai supplémentaire lors de la lecture d'une donnée. Bien entendu, cela implique que des données potentiellement corrompues par un SEU seront propagées dans le circuit. Pour pallier à ce nouveau problème, en parallèle des opérations normales effectuées par le circuit, les données doivent être vérifiées. Lorsqu'une erreur est détectée, le fonctionnement normal du circuit est stoppé en attendant que les données corrigées soient fournies par le circuit de correction. Ce qui suit dépend alors du nombre de cycles d'horloge nécessaire aux circuits de détection et de correction pour effectuer leurs tâches. Le schéma simplifié de notre solution appliquée au circuit de la figure 2-6 est donné par la figure 2-7. Dans ce cas de figure, nous avons omis dans un premier temps les autres entrées des fonctions logiques (*Entrée 1, Entrée 2 ...*). Pour faciliter l'illustration de la technique, les blocs de détection et de correction sont représentés de manière distincte, mais en réalité ils ne sont pas indépendants, puisque le bloc de correction a besoin du syndrome calculé par le bloc de

détection. Les données lues dans la mémoire sont transmises au système par l'intermédiaire d'un multiplexeur (*MUX*). L'une des entrées du multiplexeur provient directement de la mémoire, et l'autre du bloc de correction. Tant qu'aucune erreur n'est détectée, l'entrée connectée à la mémoire est directement transmise au reste du système. Lorsque le signal de détection d'erreur (*EI*) indique la présence d'une erreur, le système est stoppé pour un ou plusieurs cycles d'horloge (signal *Hold*), laissant le temps au bloc de correction de calculer les données corrigées. Une fois que les données corrigées sont disponibles la seconde entrée du *MUX* (qui provient du bloc de correction) est transmise au reste du système.

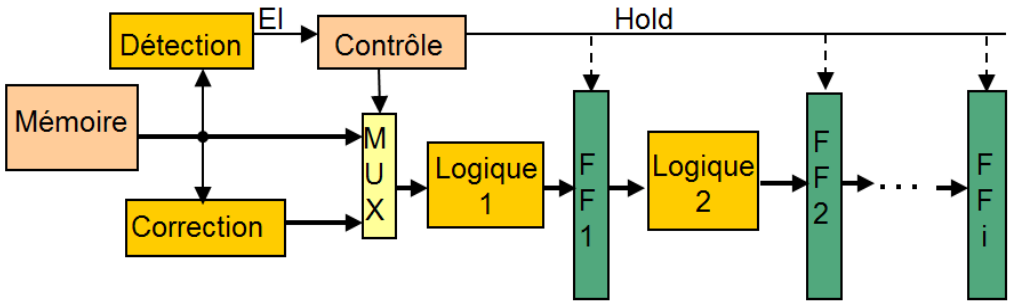

Fig. 2-7 : Architecture pour la réduction du délai du au code correcteur lors de la lecture

Cas a : Le signal de détection de l'erreur et les données corrigées sont prêtes un cycle d'horloge après que les données aient étaient lues. Le but, on le rappelle, est de fournir au système les données lues dès qu'elles sont prêtes par l'intermédiaire du *MUX*, et d'utiliser une fréquence d'horloge qui ne tienne pas compte du délai de détection et du délai de correction. Le système fonctionnera normalement, et à la bonne fréquence, tant qu'aucune erreur ne sera détectée dans les données lues en mémoire. Durant toute cette phase le *MUX* est contrôlé pour transmettre directement au système les données lues. Cependant lorsqu'une erreur est présente dans les données lues, celle-ci est transmise au reste du système et le contamine avant que le signal de détection d'erreur ne soit actif. Ici, seul *FF1* est contaminé, puisque la détection prend un cycle. Pour résoudre ce problème, lorsque le signal de détection d'erreurs *EI* indique la présence d'une erreur, il active le signal *Hold* du système pendant un cycle. Le signal bloque ainsi l'état de toutes les bascules du système sauf l'étage *FF1*. Ce signal change également la configuration du *MUX* pour qu'il transmette les données provenant du bloc de correction. Ainsi pendant un cycle d'horloge toutes les bascules vont préserver leurs états précédents sauf celles de l'étage *FF1* du circuit. Pendant ce même cycle, l'étage *FF1* est décontaminé, puisqu'il reçoit les données corrigées transmises par le *MUX*, et que les données corrigées sont disponibles. Au cycle suivant le signal *Hold* est désactivé et le circuit redémarre à partir d'un état correct (toutes les bascules contiennent une donnée valide).

L'implémentation pour le **cas a**, illustré par la figure 2-7, fonctionne tant que seules les données lues en mémoire entrent dans cet étage du circuit (entre la mémoire et *FF1*). Si d'autres entrées sont connectées à cet étage du circuit (comme le signal *Entrée 1* dans la figure 2-6), alors durant la phase de décontamination, les valeurs transmises à ces

entrées doivent être les mêmes que celles durant la phase où l'erreur a été propagée. Or entre le moment où l'erreur s'est propagée, et le moment où la décontamination s'effectue, il s'est passé un cycle d'horloge, et la valeur du signal *Entrée 1* peut être différente. Il faut donc restaurer la valeur précédente d'*Entrée 1* lors de la décontamination. Pour cela, une FIFO est ajoutée à cette entrée, pour préserver la valeur précédente durant un cycle et la transmettre au circuit par l'intermédiaire d'un multiplexeur, comme illustré par la figure 2-8. L'étage *FIFO 1* est donc composé d'un seul banc de bascule dont les valeurs sont utilisées durant les phases de correction. Le signal contrôlant le multiplexeur associé est le même que celui qui contrôle le multiplexeur transmettant les données provenant de la mémoire au reste du circuit.

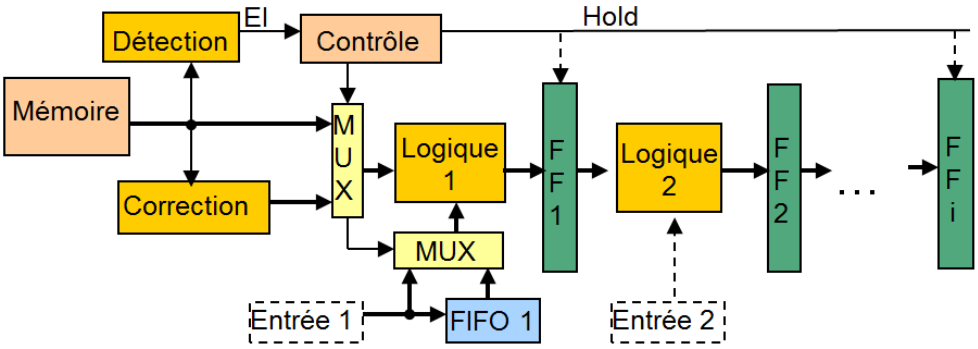

Fig. 2-8 : Suppression du délai avec sauvegarde de contexte pour un cycle d'horloge

Cas b : L'erreur est détectée en un cycle d'horloge (comme le **cas a**), et les données corrigées sont prêtes *m* cycles d'horloge après. Le circuit de correction est donc composé d'un pipeline pour effectuer le calcul des données corrigées durant *m* cycles. L'implémentation de notre solution est identique au **cas a** (Fig. 2-7). Cependant lorsqu'une erreur est détectée, le circuit est stoppé durant *m–1* cycles d'horloge. Cela laisse le temps pour les données lues en mémoire d'être corrigées. Elles sont ensuite stockées dans *FF1* durant le cycle d'horloge suivant. Durant ces *m* cycles le *MUX* transmet les données provenant du bloc de correction, et ainsi à la fin du $m^{ème}$ cycle toutes les bascules contiennent une donnée correcte. Le signal de blocage est désactivé au cycle suivant.

Cas c : Le signal de détection d'erreurs et les données corrigées sont prêts *k* cycles d'horloge après que les données aient été lues en mémoire. Les blocs de détection et de correction sont donc composés de *k* étages de pipeline. A noter qu'il faut tenir compte du délai du bloc de détection augmenté de celui nécessaire au signal *Hold* pour bloquer toutes les bascules du système. Le nombre *k* d'étages de pipeline tient compte de l'ensemble de ces délais. Dans ce cas, la décontamination du circuit prendra *k* cycles d'horloge, puisque les données contaminées ont été propagées durant *k* cycles avant que le système ne soit bloqué. L'étage 2 nécessite que l'étage 1 soit décontaminé avant d'être soi-même décontaminé. L'étage 3 doit être décontaminé après que l'étage 2 ait été décontaminé, etc… De plus, comme la correction se fait en *k* étapes une fois que les données corrigées sont prêtes, le nombre de cycles durant lesquels les bascules sont

verrouillées est variable. L'étage suivant immédiatement le circuit de correction n'a pas besoin d'être bloqué puisque les données corrigées sont prêtes; l'étage suivant doit rester bloqué pendant un cycle ; l'étage suivant ce dernier doit rester bloqué pendant deux cycles ; …; l'étage atteint *k+1* cycles d'horloge plus tard (premier étage non contaminé), ainsi que le reste du circuit sont bloqués pendant *k* cycles. Outre le fait de corriger l'état du dernier étage contaminé, l'état correct des étages débloqués au moment où le circuit a été bloqué doit être restauré (ces états sont effacés pendant la correction par le passage de la donnée corrigée). Les opérations effectuées depuis que la donnée fautive a été lue jusqu'au moment où le circuit a été bloqué doivent être ré-effectuées pendant les *k* cycles de correction.

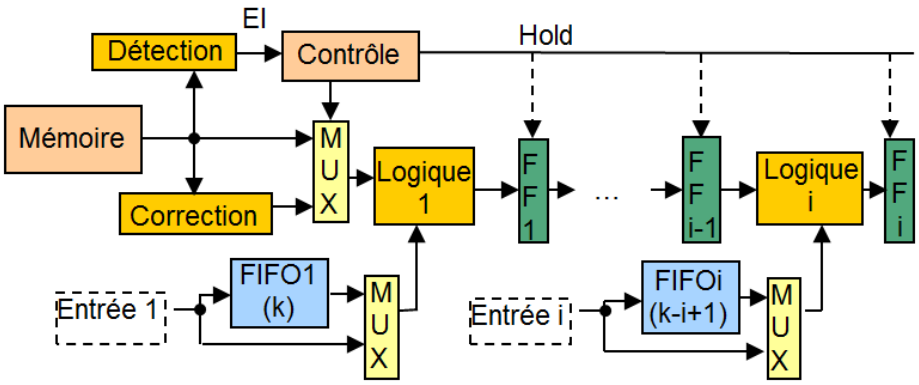

Fig. 2.9 : Architecture générale du circuit de décontamination pour k cycles d'horloge de détection

Ce principe est illustré par la figure 2-9. Les données lues en mémoire doivent donc être réémises pendant *k–1* cycles après émission de la donnée corrigée. La solution la plus naturelle consiste à utiliser les données qui sont dans le circuit de correction et de les émettre cycle après cycle. De plus, pour étendre le principe exposé dans la figure 2-8, supposons que certaines données ne provenant pas de la mémoire sont utilisées par les étages du circuit à décontaminer, (comme *Entrée 1* et *Entrée 2* dans la figure 2-7). Alors durant la décontamination, les valeurs successives appliquées à ces entrées doivent être les mêmes que pendant la phase où l'erreur s'est propagée, depuis le moment où l'erreur est entrée dans l'étage concerné par une de ces entrées, jusqu'au $k^{ème}$ cycle où le circuit a été bloqué. Ceci, comme expliqué dans le paragraphe précédent, pour restaurer l'état correct des étages précédents. Par conséquent une FIFO est ajoutée à chacune de ces entrées pour préserver ces entrées, et les restituer au circuit pendant la phase de décontamination par l'intermédiaire d'un multiplexeur. Le nombre d'étages de ces FIFO dépend de l'étage auxquels elles sont connectées. La taille de la FIFO connectée à un étage de pipeline est égale au nombre de cycles d'horloge nécessaire pour atteindre le dernier étage contaminé depuis cette FIFO. En effet, le premier étage doit ré-effectuer les *k* dernières opérations. L'étage 2 doit refaire les *k–1* dernières opérations, etc… Donc dans la figure 1, si une FIFO est nécessaire pour fournir une donnée au circuit *Logique 1*, celle-ci aura *k* étages qui sauvegarderont les *k* dernières valeurs de cette entrée. Si une FIFO est nécessaire pour fournir une donnée au circuit *Logique 2*, celle-ci aura *k–1* étages qui sauvegarderont les *k–1* dernières valeurs de cette entrée.

A retenir : Plus généralement si une FIFO est nécessaire pour fournir une donnée à l'$i^{ème}$ étage contaminé, elle aura $k–i+1$ étages. La *FIFO i* est activée à chaque cycle pendant les opérations normales et pendant la décontamination, à part lorsque l'étage *FFi* est bloqué : à ce moment elle préserve les données stockées.

Cas d : Le signal de détection est prêt k cycles d'horloge après que les données aient été lues, et les données corrigées m cycles d'horloge après la lecture. ($m>k>1$). L'implémentation de la solution est identique au **cas c** (y compris la taille des FIFO), excepté pour le nombre de cycles d'horloge pendant lesquels les étages sont verrouillés. Dans le cas présent les données sont prêtes $m–k$ cycles d'horloge après le blocage du circuit. Par conséquent tout le circuit est bloqué pendant $m–k$ cycles, le temps que les données corrigées soient prêtes, avant de commencer la décontamination de la même manière que dans le **cas c**.

2.6 Adaptation de la solution pour les architectures irrégulières

L'approche présentée dans la section précédente est applicable directement dès l'instant où l'architecture visée est *régulière*. Pour notre approche, régulière, signifie que la sortie de la mémoire est connectée à un pipeline régulier tel que présenté dans les figures précédentes. Moyennant toutefois quelques ajustements, il est possible d'adapter cette solution à n'importe quelle architecture synchrone. Les différents cas de figure sont présentés ci-après.

2.6.1 Ressources partagées par plusieurs étages

Une FIFO additionnelle peut être connectée à plusieurs entrées relatives à différents étages du pipeline. Dans ce cas, si la FIFO est activée lorsque l'étage relatif à la première entrée est débloqué, le second étage n'est pas activé. Cela ne constitue pas un problème supplémentaire, et la donnée qui doit être transmise au second étage le sera au moment ad hoc. En effet, considérons l'exemple de la figure 2-10. La FIFO de taille 2 est connectée au premier et au second étage, le premier étage aura besoin des deux états stockés dans la FIFO lors de la décontamination. En revanche, le second étage n'a pas besoin de la première des deux données. En effet, lorsque le premier étage utilise la première des deux données, cela correspond au cycle où la donnée corrompue était stockée dans *FF1*. Durant ce cycle, l'étage 2 n'a pas été contaminé et n'a pas besoin de cette donnée qui était appliquée à son entrée durant ce cycle. Au cycle suivant l'étage 1 a besoin de la donnée suivante pour poursuivre la décontamination, et l'étage 2 également pour effectuer la décontamination. Les données devant servir à la décontamination des deux étages sont identiques au moment requis. Cet exemple peut être généralisé à n'importe quel cas de figure du même type.

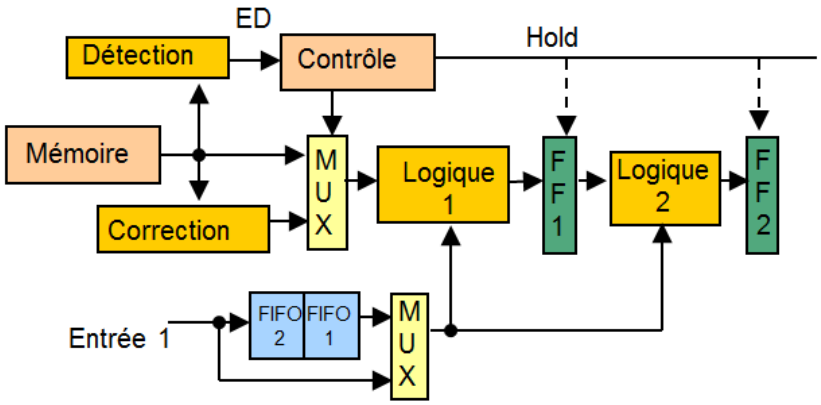

Fig. 2-10 : Partage de ressources entre plusieurs entrées d'un même pipeline

2.6.2 Ressources partagées par plusieurs mémoires

Considérons un circuit utilisant plusieurs mémoires protégées par un code correcteur. Deux cas de figure sont possibles. Considérons deux mémoires pour simplifier ; ces deux mémoires utilisant la solution proposée, en cas d'erreur, propagent une donnée erronée pendant un certain nombre de cycles à l'intérieur du circuit. Si les deux chemins sont entièrement indépendants, il n'y a aucune contrainte au niveau de l'implémentation de la solution. A l'inverse, considérons une donnée commune aux deux chemins de deux mémoires (*Entrée 1*). A noter que cette entrée peut elle-même provenir d'une troisième mémoire. Cette donnée peut-être utilisée à deux instants différents, suivant que la donnée corrigée provient de l'une ou l'autre des mémoires. L'une des raisons est que le code correcteur utilisé pour l'une de ces mémoires nécessite un temps de détection moins long que l'autre. L'autre raison est illustrée par la figure 2-11. Dans cette figure on a volontairement omis les circuits de détection-correction-contrôle, afin de ne pas la surcharger.

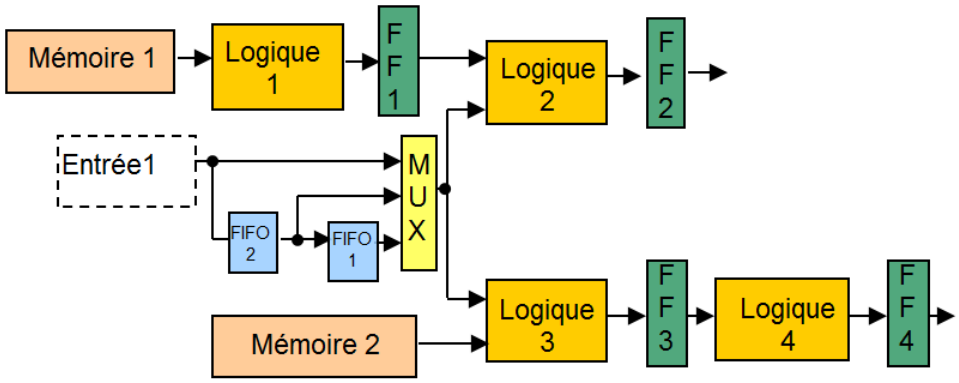

Fig.2-11 : Partage de ressources entre plusieurs chemins contaminés

Dans cette figure la donnée d'*Entrée 1* doit être disponible dès le premier cycle de correction pour la *Mémoire 2*, et seulement à partir du second cycle de correction pour la *Mémoire 1*. Considérons que le temps de détection de l'erreur est de deux cycles pour chacune des deux mémoires, la décontamination, lorsque la donnée provient de la *Mémoire 1*, a besoin uniquement de la dernière donnée provenant de l'*Entrée 1*. En revanche lorsque la donnée à corriger provient de la *Mémoire 2*, la décontamination a

besoin des deux dernières données. Une FIFO de taille deux doit donc être associée à l'*Entrée 1* (on considère toujours le cas le plus défavorable). Quand la correction se fait sur le chemin provenant de la *Mémoire 2*, il n'y a aucun changement dans le processus, *FF1* et *FF2* étant bloquées. Quand la donnée corrigée provient du chemin issu de la mémoire 1, il faut soit récupérer directement la donnée stockée dans le deuxième étage de la FIFO (illustré par la figure 2-11), soit activer la FIFO dès le premier cycle de la décontamination, afin que la donnée désirée soit la donnée fournie par la FIFO au second cycle. Dans ce cas au premier cycle la donnée de *FIFO 2* est transmise à *FIFO 1*, et cela sans changer les états de *FF2* et *FF3* puisque seul *FF1* est débloqué. La solution est rigoureusement équivalente au cas énoncé plus haut où les temps de détection ne sont pas identiques pour deux codes correcteurs protégeant deux mémoires distinctes.

2.6.3 Chemins contaminables de taille différente

Un autre problème du à l'architecture irrégulière des pipelines, est illustré dans la figure 2-12. Dans cette figure nous supposerons dans un premier temps que l'erreur est détectée en deux coups d'horloge. Au premier cycle de propagation de l'erreur, *FF1* est contaminée, au second cycle, *FF2* et *FF3* sont contaminées. Lors de la décontamination *FF2* et *FF3* seront donc décontaminées en même temps. Or pour être décontaminée, *FF3* a besoin de l'état valide précédent de *FF2* qui a été effacé lors du dernier cycle de contamination. Ce dernier état doit donc être sauvegardé dans une FIFO qui sera utilisée lors du second cycle de décontamination. Supposons maintenant, toujours avec la même architecture, que le temps de détection soit de trois cycles. Au deuxième cycle de décontamination, *FF3* à besoin pour être décontaminée de l'état valide de *FF2* au second cycle de contamination. Cet état valide existait au deuxième cycle de contamination, donc il faut restaurer l'état, deux cycles avant le blocage du circuit. Conformément à ce qui a pu être dit (section 2.5, cas c) une FIFO de taille deux doit stocker les deux derniers états de *FF2*, comme si on considérait la donnée fournie par *FF2* comme une source externe. En revanche, le deuxième étage de la FIFO contient la donnée contaminée, émise par *FF2* lors du troisième cycle de contamination. Par conséquent il ne faut pas la réintroduire dans le circuit principal. Or *FF2* a également été décontaminée lors du deuxième cycle de décontamination. Donc lors du troisième cycle de décontamination, au lieu d'utiliser la donnée qui se trouve maintenant dans le premier étage de la FIFO, et qui est contaminée, on utilisera la donnée provenant de *FF2* qui est maintenant valide.

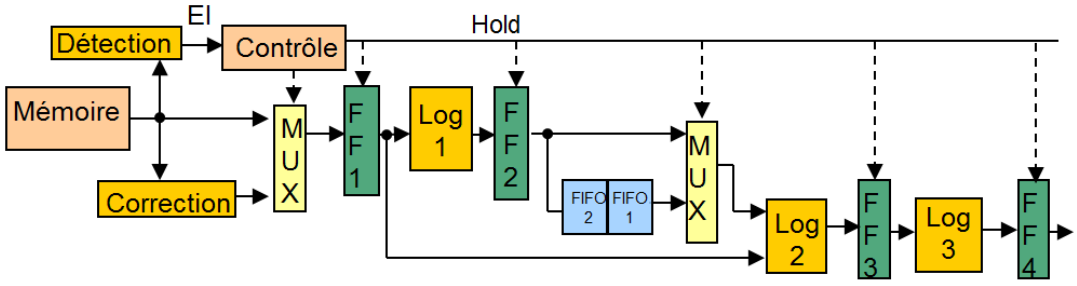

Fig. 2-12 : Rajout de ressources dans un chemin contaminé

Généralisation : Ce principe peut se généraliser à n'importe quel circuit comportant une fonction logique située au confluent de plusieurs chemins contaminables. Si ces chemins sont de tailles différentes, le plus court ne nécessite pas de FIFO additionnel. Si ce composant est atteint depuis la mémoire en l_s cycles d'horloge, alors toutes les flip-flops connectées en entrée doivent fournir un état valide lors du $l_s^{ème}$ cycle de décontamination. Toutefois, supposons que l'une des flip-flops connectées en entrée de cette fonction soit atteinte depuis la mémoire en un nombre de cycle d'horloge l_f, tel que $l_s \leq l_f \leq k$. Dans ce cas, cette flip-flop n'aura pas été décontaminé avant le cycle pour lequel son état correct est requis. On doit donc lui associer une FIFO de taille $k - l_s + 1$ qui fournira une donnée correcte aux cycles de décontamination l_s, l_s+1, ... l_f. Vu que $l_s \leq l_f$, ces données ne sont pas contaminées. Au cycle l_f+1, la flip-flop aura été décontaminée, et fournira une donnée correcte pour la décontamination.

Une autre solution est présentée par la figure 2.13. Dans cette figure, on a déplacé les FIFO destinées à sauvegarder les deux derniers états de *FF2* en sortie du circuit de détection/correction. En temps normal cela reviendrait à ajouter en sortie de la mémoire une FIFO de taille 4. En effet, pour restaurer l'état de *FF2* deux cycles avant l'interruption, il faut restaurer l'état de *FF1* trois cycles avant l'interruption. Pour restaurer l'état de *FF1* trois cycles d'horloge avant la détection de l'erreur, il faut réémettre la donnée émise par la mémoire quatre cycles avant.

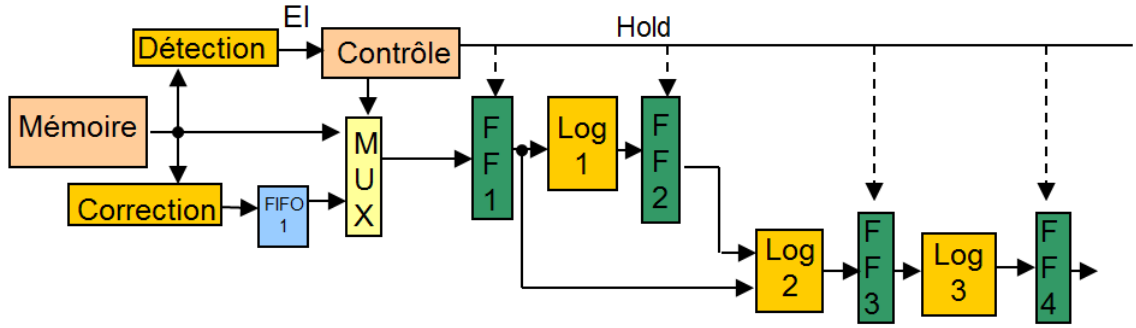

Fig. 2-13 : Rajout de ressources dans un chemin contaminé : deuxième solution

A retenir : A chaque fois que l'on recule d'un étage de pipeline la position d'une FIFO destinée à sauvegarder les états précédents, le nombre d'étages de la FIFO augmente en rapport d'une unité.

Toutefois, comme le circuit de détection/correction est au global lui-même composé d'au moins trois étages de pipeline, (la détection se faisant en trois cycles d'horloge), dans le cas le plus défavorable il suffit de rajouter un étage de pipeline en sortie du bloc de détection/correction pour restaurer les quatre dernières données émises par la mémoire. L'avantage est que l'on réduit le matériel supplémentaire pour effectuer la décontamination. L'inconvénient est que la décontamination augmente d'un cycle d'horloge. En effet les quatre dernières données lues en mémoire devant être réémises, la décontamination commence un cycle avant que la donnée corrigée ne soit émise. Durant

le premier cycle la première donnée stockée dans l'étage de la FIFO supplémentaire est émise, alors que *FF1* est débloquée. Au deuxième cycle, *FF2* est débloquée (pas *FF3*), et l'état de *FF2* deux cycles avant le blocage de l'horloge est restauré. Durant le même cycle, la donnée corrigée est émise. Au troisième cycle *FF3* peut être décontaminée, et *FF4* au quatrième cycle.

A retenir : Il peut donc être utile de réémettre des données lues pendant plus de k cycles. Par conséquent, si dans un premier temps l'utilisation des données présentes dans le circuit de détection-correction suffit, dans le cas général il faut rajouter une FIFO en sortie du bloc de correction pour compenser la différence entre le nombre de cycles de décontamination et le nombre *k*.

2.6.4 Architecture comportant une boucle

La figure 2-14 est le schéma d'un circuit comportant une boucle dans un chemin contaminable. Considérons un code correcteur avec un temps de détection de deux cycles. Comme dans l'exemple précédent, pour la décontamination de *FF2*, l'état de *FF1*, deux cycles avant le blocage, doit être restauré. De la même manière que précédemment, le deuxième étage de la FIFO ne doit pas être utilisé, ce qui ne présente pas d'inconvénient majeur, car *FF1* vient d'être décontaminée durant le cycle précédent. De ce point de vue la méthode pour décontaminer ce circuit est également la même que dans la figure 2-12. La principale différence avec le cas précédent, est que la FIFO additionnelle, associée à *FF1*, ne présente pas d'intérêt à être déplacée à un autre endroit du circuit. *FF1* a besoin de la donnée corrigée de la mémoire et aussi de son propre état valide deux cycles auparavant. Par conséquent, même si comme dans le cas précédent on prenait un cycle d'horloge supplémentaire pour effectuer la décontamination, il faudrait maintenir une FIFO associée à *FF1*, qui aurait un étage de plus.

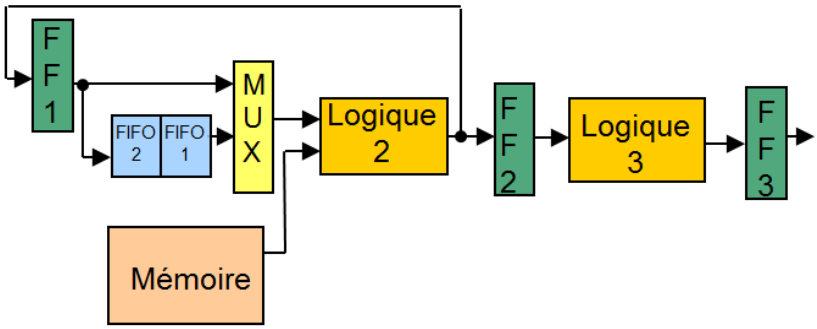

Fig. 2-14 : Rajout de ressources dans le cas d'une boucle

2.7 Pertes de performances dues à la solution

L'approche proposée permet d'implémenter un code correcteur dans une mémoire embarquée, sans affecter la fréquence du circuit, ni introduire des étages de pipeline supplémentaires, en dehors des circuits de décontamination parallèles présentés. En effet durant le fonctionnement normal du circuit, c'est-à-dire tant qu'aucune erreur n'est

détectée, le circuit fonctionnera avec une fréquence identique à celui du même circuit non protégé par un code correcteur, et sans ajouter des cycles d'horloge pour lire une donnée en mémoire. En revanche, chaque fois qu'une erreur sera détectée, un ou plusieurs cycles d'horloge seront nécessaires pour effectuer la décontamination. Toutefois cette perte est négligeable. Par exemple, si nous considérons un circuit comportant 60Mbytes de mémoire embarquée, et un SER de 1000 FIT par Mbit, le nombre total d'erreurs par heure sera de $60\times8\times1000\times10^{-9} = 48\times10^{-5}$, soit une erreur présente en mémoire tous les 87 jours (1 FIT = 10^{-9} erreur par heure). Ceci est totalement inacceptable dans un certain nombre d'applications tels que les réseaux, les serveurs, l'automobile, etc… et impose l'utilisation de codes correcteurs pour protéger les mémoires embarquées. Comme l'utilisation de codes correcteurs peut en revanche provoquer une perte de performance inacceptable en termes de fréquence de fonctionnement, notre approche peut être utilisée pour supprimer cet inconvénient. Considérons maintenant un nombre de cycles d'horloge relativement grand pour la détection d'erreurs, (trois cycles d'horloge) l'utilisation de notre approche induit la perte de 3 cycles (ou un peu plus le temps que les données corrigées soient prêtes) tous les 87 jours. Alors pour une fréquence d'horloge de 200 MHz cela provoque une baisse de performance de 10^{-15} %, ce qui est insignifiant.

2.8 Cas d'étude et résultat

Le cas d'étude consiste en un modulateur OFDM [MJ05] [Sah09], proposé dans le cadre de la radio logicielle. Le but de la radio logicielle est de pouvoir supporter plusieurs standards de communication avec un seul émetteur/récepteur. La modulation OFDM est particulièrement importante dans le développement de la radio logicielle, à cause du grand nombre de standard utilisant ce type de modulations (B3G, 4G, WIFI, WIMAX, WRAN). L'architecture considérée ici est capable de prendre en main plusieurs modulations QAM et OQAM. Ces modulations sont basées sur des FFT direct (réception) et inverse (émission), utilisant les algorithmes de type radix, ainsi que le filtrage IOTA pour la modulation OQAM. IOTA est l'acronyme de "Isotropic Orthogonal Transform Algorithm pulse shaping filtering". Le filtrage par la fonction IOTA améliore les performances contre les interférences inter-porteuses, le bruit impulsionnel, et les canaux à évanouissement [MJ05]. Par conséquent, la modulation OQAM ne nécessite pas d'intervalle de garde additionnel pour avoir les mêmes performances que la modulation QAM classique. En revanche, les symboles OFDM doivent être calculés deux fois plus vite pour conserver le même débit global. Le circuit peut supporter un nombre variable de sous-porteuse de 64 à 8192, et peut effectuer jusqu'à 4 modulations en parallèle. C'est une architecture à base de mémoire, et donc leur protection est fortement recommandée.

Cette architecture (Fig. 2-15) est composée de 2 mémoires ROM qui stockent les coefficients pour le calcule de la FFT et le filtrage IOTA. 2 mémoires RAM stockent les échantillons et résultats intermédiaires des opérations de FFT et de filtrage. La mémoire RAM dédiée à la FFT est divisée en deux blocs qui sont utilisés alternativement. Chacun de ces blocs est divisé en 8 sous blocs qui stockent les échantillons complexes. On peut

donc considérer, en séparant les parties imaginaires et réelles que chaque bloc possède 16 ports d'entrées et 16 ports de sorties. Chacune de ces 16 mémoires est un bloc reconfigurable en fonction du nombre de sous-porteuses et de modulations en parallèle. La matrice de calcul est aussi reconfigurable, divisée en 12 blocs de calcul multiplication/accumulation (Fig. 2-16). La configuration de la matrice de calcul dépend de l'algorithme choisie (radix 2, 4 or 8), du nombre de modulation en parallèle et si le filtrage par la fonction IOTA est effectuée ou non. Les blocs de calcul qui ne sont pas utilisé sont désactivés. Des registres peuvent également être désactivés dans les blocs de calcul actifs en fonction du mode de calcul.

Les résultats présentés ici concernent la protection des mémoires RAM stockant les échantillons utilisées dans les opérations de FFT. Comme les données peuvent être écrites et lues en mémoires en même temps, on utilise des circuits de codage et décodage séparés. Cependant le bloc de contrôle est commun à toutes les entrées/sorties pour simplifier le contrôle de l'architecture en cas d'erreur. Il peut y avoir selon les configurations une différence de phase entre les horloges de π par rapport au reste du circuit. Par conséquent, si on utilise des codes correcteurs de façon directe, on dispose alors de moins d'un demi-cycle d'horloge pour la détection et la correction d'erreur. Même pour les codes les plus simples, ce temps est insuffisant, et demandera de réduire la fréquence d'horloge ou d'ajouter des étages de pipeline. Dans les deux cas nous avons une réduction du débit d'autant plus importante que, suivant les configurations, les données doivent effectuer plusieurs boucles entre la matrice de calcul et la mémoire. Donc nous avons utilisé l'approche décrite dans ce chapitre pour éviter les pertes de performances.

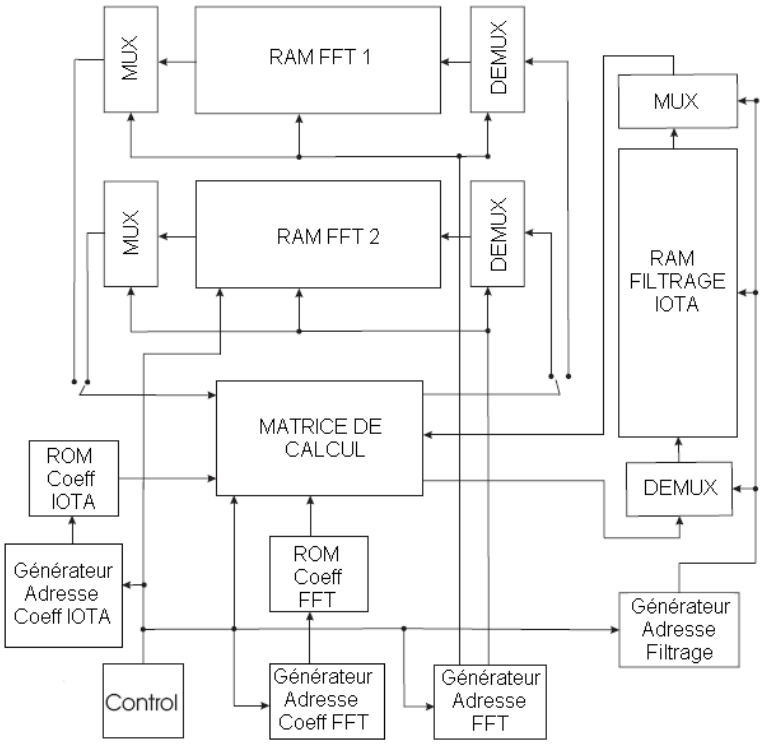

Fig. 2-15 : Architecture du modulateur OFDM avancé

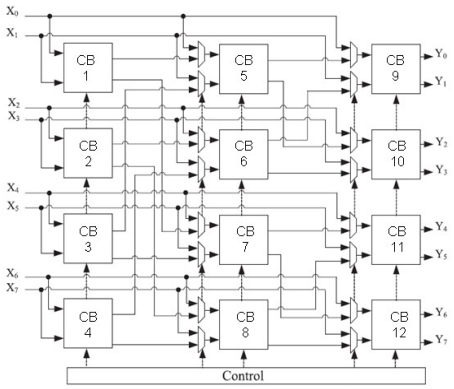

Fig. 2-16 : Matrice de calcul reconfigurable du modulateur

La tâche la plus complexe pour implémenter cette approche est d'analyser l'architecture complexe du modulateur OFDM reconfigurable, et de déterminer où doivent être placé dans le chemin de décontamination les FIFO et multiplexeur. Comme nous l'avons vue dans la section précédente, la longueur du chemin de décontamination et la taille des FIFO dépend du délai du bloc de détection. L'analyse varie donc d'un code à un autre. La présentation détaillée est très longue et va de pair avec la présentation détaillée de l'architecture du circuit. Par conséquent la suite présente uniquement les principes généraux. Les deux blocs de mémoire RAM dédiée au calcul de la FFT étant utilisés alternativement, un seul système de codage-décodage protégeant l'ensemble des entrées sorties englobant les deux mémoires est suffisant. Ce système englobe également les MUX/DEMUX de ces mémoires (Fig. 2-15). Ces composants répartiront ensuite les données dans les mémoires stockant les bits de données et les bits de contrôle correspondantes. Dans chaque cas il faut bien entendu tenir compte de toutes les configurations possibles de la matrice de calcul. Ainsi que nous l'avons dit, chaque bloc de calcul dans la matrice peut être désactivé en fonction de l'algorithme exécuté en cours. Par conséquent si les premiers étages sont impliqués dans la correction (CB 1, 2, 3 and 4), les autres blocs peuvent être stoppés. Si c'est l'étage du milieu qui est concerné par la correction (CB 5, 6, 7 and 8), le premier étage est déjà désactivé, et le troisième doit être stoppé. En les désactivant alors qu'ils contiennent des données non corrompu, on préserve ainsi un état correct à l'intérieur des blocs de calcul. Toutefois à cause de leur complexité, une description un peu plus détaillée de ces blocs lorsqu'ils sont activés est nécessaire, au moins pour les plus compliqués d'entre eux.

Les plus compliqués ont une architecture composée en partie par trois étages de pipeline avec une boucle entre le premier et le dernier étage. Par conséquent des FIFO et des MUX additionnels sont nécessaire pour sauvegarder le contexte dans ce pipeline (Fig.2-17). Ici nous avons positionné les FIFO en entrée du pipeline, après les multiplexeurs situés avant le premier étage. Normalement, pour respecter scrupuleusement les principes présentés dans la section 2.6, ces FIFO auraient dû être situées en sortie du dernier étage, avant les multiplexeurs. Mais comme dans nos expérimentations, aucun étage au-delà de cette partie du circuit n'était contaminé, cela n'était pas nécessaire.

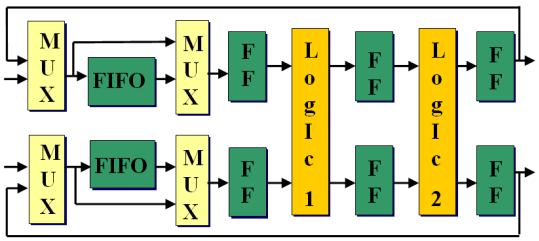

Fig. 2-17 : Architecture des modules de calculs avec sauvegarde de contexte

Le circuit avec plusieurs implémentations de codes correcteurs a été synthétisé avec une technologie 65 nm grâce au logiciel Design Vision, et simulé avec Modelsim. La fréquence d'horloge originale était de 278 MHz (i.e. sans ajouter de code correcteur). Plusieurs codes mono et multi correcteurs ont été implémentés. Hamming étendu (i.e. avec bit de parité) (22, 16, 1), Hsiao (22, 16, 1), BCH (31, 16, 3), Reed-Solomon (56, 16, 17), ainsi que Reed-Solomon (20, 8, 4). Entre parenthèse sont donnés, pour chaque code, en premier le nombre total de bits (bits de donnés + bits de contrôle), le deuxième nombre donne le nombre de bits de donnés, et en dernier le nombre d'erreur corrigeable. Les données dans l'architecture sont codées sur 16 bits. Les 8 bits dans le code Reed-Solomon (20, 8, 4) implique que les 16 bits sont divisés en deux partie, chacune protégée par un code RS (20, 8, 4).

Le tableau 2-1 présente les résultats pour différentes implémentations. La première colonne présente le code évalué, la deuxième présente la fréquence de fonctionnement maximum atteinte avec une implémentation standard des codes (sans technique de réduction des délais), et la troisième colonne la fréquence que l'on a pu atteindre avec l'implémentation proposée. Les colonnes 4 et 5 donnent les pénalités de surface et de consommation dues à l'implémentation proposée, en comparaison avec l'implémentation standard. Ces valeurs ont été calculées par rapport à la mémoire protégée par un code correcteur implémenté de façon standard et non par rapport à la surface et la puissance consommée par l'ensemble du circuit. Ces coûts en surface et en consommation auraient été plus faibles s'ils avaient été calculés par rapport à l'ensemble du circuit. Mais en procédant ainsi, même s'ils dépendent des FIFO et multiplexeurs rajoutés, ces résultats auraient été moins significatifs. En effet, ils varieraient en fonction des circuits protégés, et de l'importance des mémoires par rapport au reste du circuit. De plus, ces coûts auraient été plus faibles, si à taille de données constantes (16 bits), le nombre de mots stockés par mémoire avait été plus important. Dans tous les cas la fréquence de fonctionnement atteinte grâce à notre solution est égale à la fréquence de fonctionnement originale du circuit. Nous avons constaté une augmentation drastique de la vitesse de calcul par rapport à l'implémentation de base des différents codes. (40% pour Hamming, 42% pour HSIAO, et jusqu'à 329% for Reed-Solomon (56, 16, 17)), alors que leur surfaces et leurs consommations additionnelles respectives sont faibles. En effet pour une mémoire de 512 kbits, la surface supplémentaire due à l'utilisation de notre solution représente moins de 3% de la surface initiale de la mémoire protégée, avec une surconsommation en énergie électrique d'environ 6%. A noter que le fait de calculer le

surcoût par rapport à la mémoire protégée, implique que ce surcoût relatif est plus important dans les cas des codes de Hamming et de Hsiao (pourtant plus simples), que dans le cas de codes plus complexes.

Tab. 2-1: Résultats de l'implémentation de notre solution dans le modulateur OFDM pour les mémoires RAM-FFT

Mesure / Code	Fréquence standard	Fréquence sans délai	Pénalité de surface	Pénalité de consommation
Hamming (22,16)	198 MHz	278 MHZ	+2,64%	+6,01%
Hsiao (22, 16)	196 MHz	278 MHZ	+2,69%	+3,62%
BCH (31, 16, 3)	151 MHz	278 MHZ	+2,15%	+9,04%
RS (56, 16, 17)	64,7 MHz	278 MHZ	+2,24%	+8,27%
RS (20, 8, 4)	171 MHz	278 MHZ	+1,91%	+6,75%

Concernant la pénalité de temps qui existe encore en cas de détection d'erreur, pour les codes de Hamming et de Hsiao, les données contaminées sont propagées pendant un cycle avant la détection de l'erreur et l'activation du signal de blocage. Par conséquent la décontamination se fait ensuite en un cycle d'horloge. Pour les codes BCH (31, 16, 3) et RS (20, 8, 4), les données erronées sont propagées pendant deux cycles avant que le circuit ne soit interrompu. Par conséquent la décontamination se fait sur deux cycles d'horloge. Enfin pour le code RS (56, 16, 17), jusqu'à trois cycles d'horloge peuvent être nécessaire avant de stopper le fonctionnement normal, et donc trois cycles seront également nécessaire pour la décontamination. De plus un cycle d'horloge supplémentaire est nécessaire avant de disposer de la donnée corrigée. Mais ainsi que nous l'avons souligné dans la section précédente, cela représente une pénalité de temps insignifiante même pour les codes les plus complexes. Ce qui veut dire que, si on accepte la pénalité de surface et de consommation dues à l'implémentation de codes complexes, ceux-ci peuvent être implémentés sans réduire le débit original du circuit.

2.9 Conclusion

En faisant subir un traitement différent aux bits de données et aux bits de contrôle, on peut supprimer la pénalité de temps dû à l'utilisation d'un code correcteur lors de l'écriture d'une donnée en mémoire. En transmettant les données aux modules de calcul avant de détecter une erreur, on supprime la pénalité de temps lors de la lecture d'une donnée en mémoire. La seule pénalité qui subsiste est constituée par les quelques rares cycles supplémentaires pour effectuer la décontamination quand une donnée erronée est propagée dans le circuit avant que l'erreur ne soit détectée. Cette pénalité représente une perte de performance insignifiante et est négligeable. Le circuit pour effectuer la décontamination est constitué, outre les circuits de détection et de correction, par des

FIFO et des multiplexeurs supplémentaires. Ce matériel est destiné à rétablir le contexte du circuit lorsque l'erreur s'est propagée pour effectuer une décontamination correcte. Dans le cas où la mémoire protégée est connectée à un pipeline régulier, ce matériel est positionné à chacune des autres entrées du pipeline. Dans le cas des architectures plus complexes, il peut être nécessaire de rajouter des composants à l'intérieur des modules de calculs. Au final cela permet de maintenir la fréquence et le débit original d'un circuit avec une pénalité de surface et de consommation relativement faible. Les expérimentations donnent un surcoût en matériel de 3% et environ 6% de consommation électrique supplémentaire. Toutefois l'implémentation de cette solution nécessite une étude détaillée de l'architecture de chaque circuit qui peut être difficile dans certains cas. En particulier certaines architectures peuvent comporter par un grand nombre de mémoires. Déterminer l'implémentation de la solution proposée dans ce chapitre peut être long et fastidieux pour certains circuits. Le chapitre suivant propose justement une méthode pour automatiser cette implémentation.

Chapitre 3. Formalisme et Algorithme

Le chapitre précédent présentait la solution au niveau RTL permettant de limiter le délai dû à l'utilisation de codes correcteurs. Cependant, leur implémentation systématique dans n'importe quel circuit peut être complexe. Ce chapitre détaille les principes qui permettent d'automatiser l'implémentation. L'implémentation pour la partie "écriture en mémoire" de la solution ne dépend pas de l'architecture considérée. Elle ne présente donc pas de difficulté particulière, et ne sera pas traitée dans ce chapitre. L'implémentation de la solution pour la partie "lecture en mémoire" dépend de l'architecture et du nombre de cycles d'horloge k avant la détection de l'erreur. Le chapitre précédent, et les différents schémas qui s'y rapportent ont montré qu'une connaissance très détaillée du fonctionnement du circuit n'est pas indispensable. Pour représenter ce circuit nous utiliserons un graphe orienté, dont les nœuds représentent les composants de stockage, et les arcs leurs interconnexions, en même temps que le sens de propagation des données. A partir de cette représentation il est possible de déterminer les FIFO à ajouter pour implémenter la solution, avec la possibilité de minimiser le surcoût matériel. L'algorithme créé a été testé sur plusieurs cas d'étude montrant l'importance de l'architecture dans l'implémentation.

3.1 Introduction aux graphes

La théorie des graphes a vu le jour en 1736 : le mathématicien allemand L. Euler donna ainsi une réponse au problème des *sept ponts de la ville de Koenigsberg,* à savoir : *comment traverser les sept ponts de cette ville sans jamais passer deux fois par le même*. La théorie des graphes a connu son essor après la deuxième guerre mondiale, surtout dans le cadre de la modélisation de problèmes concrets. Ils interviennent naturellement dans la représentation de réseaux de transport ou de télécommunication, par exemple, mais également dans la représentation de structures relationnelles abstraites.

Mathématiquement, un graphe G prend la forme $G = (V, A)$ où V est un ensemble de nœuds (ou sommets) et $A \subseteq V \times V$ un ensemble d'arêtes. Une arête a de l'ensemble A est définie par une paire **non-ordonnée** de nœuds, appelés les extrémités de a. Si l'arête a relie les nœuds u et v, on dira que ces nœuds sont **adjacents**, ou **incidents** avec a, ou encore que l'arête a est **incidente** avec les sommets u et v.

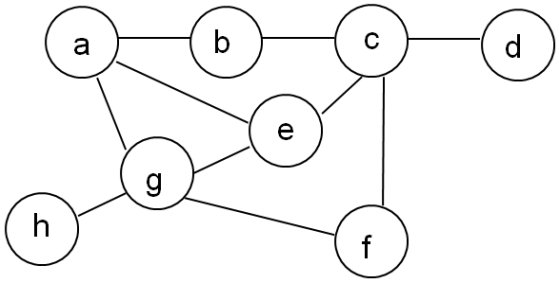

Fig. 3-1 : Exemple de graphe non-orienté

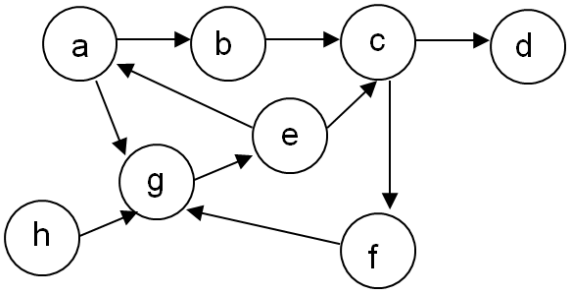

Fig. 3-2 : Exemple de graphe orienté

En donnant un sens aux arêtes d'un graphe, on obtient un digraphe (ou graphe orienté). Un digraphe $G = (V, A)$ est un ensemble de nœuds (ou sommets) V, et un ensemble d'arcs A. Un arc a de l'ensemble A est défini par une paire **ordonnée** de sommets. Lorsque $a = (u, v)$, on dira que l'arc a va de u à v. On dit aussi que u est l'extrémité initiale et v l'extrémité finale de a. Contrairement aux arêtes d'un graphe non orienté il n'y a pas de réciprocité. Dans la figure 3-1 l'arête *(a, b)* permettait d'aller aussi bien du nœud a au nœud b que l'inverse. Dans la figure 3-2 ci-après en revanche, l'arc *(a, b)* ne permet pas d'aller du nœud b au nœud a.

C'est cette dernière représentation que nous allons utiliser par la suite. En effet un graphe orienté permet de modéliser les interconnexions entre les composants de stockage (mémoire, registre, bascule), mais également le sens de propagation des données. Ainsi deux composants de stockage *u* et *v*, sont reliés par un arc *(u, v)* si et seulement si il existe un chemin combinatoire reliant la sortie de *u* à l'entrée de *v*. On peut aussi dire, toujours par analogie aux circuits électroniques, que le passage de u à v se fait en 1 cycle d'horloge. Les principales notations et fonctions utilisées dans ce chapitre sont détaillées dans la section suivante.

3.2 Etude théorique du problème

Comme on l'a rappelé au début de ce chapitre, une connaissance très détaillée du fonctionnement du circuit que l'on souhaite protéger en utilisant notre solution n'est pas indispensable. Ainsi le circuit va être modélisé par un graphe orienté, dont les nœuds représentent les composants de stockage, et les arcs leurs interconnexions, en même temps que le sens de propagation des données. Cette section présente les notations utilisées et la façon dont le circuit est décomposé en plusieurs ensembles pour déterminer l'implémentation de notre solution.

3.2.1 Définitions

N : est l'ensemble des entiers naturels {0, 1, 2 …} ; $\mathbb{N}^*$ désigne le même ensemble, privé de l'élément 0 : {1, 2 …}.

Card : La fonction Cardinal donne le nombre d'éléments qui compose un ensemble. Exemple : *Card({a, b, c}) = 3*.

V (Nœud) : L'ensemble *V* est composé de tous les composants de stockage (registre ou bascule isolée) ainsi que la sortie des mémoires.

A (Arcs) : Soit $(v_1, v_2) \in V^2$, si la sortie de v_1 est connectée au travers d'un chemin combinatoire à l'entrée de v_2, alors (v_1, v_2) est un arc et ***A*** l'ensemble des arcs du circuit.

G(V,A) : La représentation d'un circuit séquentiel est notée ***G(V, A),*** *V* étant l'ensemble des nœuds et *A* celui des arcs. La figure 3-2 représente un exemple de graphe avec *V={a, b, c, d, e, f, g, h}* et *A ={(a, b), (a, g), (b, c), (c, d), (c, f), (e, a), (e, c), (f, g), (g, e), (h, g)}*.

$\mathbf{P_n}$ (Chemin d'ordre *n*) : Soit $n \in \mathbb{N}^*$ et $(v_0, \ldots v_n) \in V^{n+1}$, tels que $\forall i, 0 \leq i < n, (v_i, v_{i+1}) \in A$. Alors $(v_0, \ldots v_n) \in \mathbf{P_n}$. On dit que $(v_0, \ldots v_n)$ est un chemin allant de v_0 à v_n de longueur *n* (ou d'ordre *n*). On peut aussi dire que la distance pour aller de v_0 à v_n par ce chemin est de *n*. Dans la figure 3-2 il y a plusieurs chemins pour relier *h* à *d*, comme *(h, g, e, c, d)* qui est un chemin d'ordre 4, ou encore *(h, g, e, a, g, e, a, b, c, d)* qui est un chemin d'ordre 9.

AP_n (Chemin acyclique d'ordre n) : Soit $n \in \mathbb{N}^*$ et un chemin $(v_0, \ldots v_n) \in \boldsymbol{P_n}$ tels que : $\forall (i, j),\ 0 \leq i < j \leq n,\ v_i \neq v_j$. Alors $(v_0, \ldots v_n) \in \boldsymbol{AP_n}$. On dit que $(v_0, \ldots v_n)$ est un chemin acyclique car la propriété supplémentaire par rapport aux éléments de P_n implique de ne jamais passer deux fois par le même nœud. Ainsi dans la figure 3-2, s'il existait une infinité de chemins reliant h à d, il n'y a que deux chemins acycliques les reliant : (h, g, e, c, d) et (h, g, e, a, b, c, d).

d_{min} (Distance minimale) : Soit $(u, v) \in V^2$, tels qu'il existe $n \in \mathbb{N}^*$ et $(v_0, \ldots v_n) \in \boldsymbol{P_n}$ avec $v_0 = u$, et $v_n = v$. Alors $d_{min}(u, v) = min\{ n \in \mathbb{N}^* / \exists (v_0, \ldots v_n) \in \boldsymbol{P_n},\ v_0 = u,\ et\ v_n = v \}$. Dans la figure 3-2 $d_{min}(h, d) = 4$.

ATTENTION. Il est important de noter que la fonction $\mathbf{d_{min}}$ est une fonction de $V \times V \rightarrow \mathbb{N}^*$. Autrement dit la distance '0' n'est pas définie, même si les nœuds u et v peuvent être identiques. Par exemple dans la figure 3-2, $d_{min}(e, e) = 3$. En revanche $d_{min}(h, h)$ n'existe pas. De plus $d_{min}(a, b) = 1$, alors que $d_{min}(b, a) = 6$. En conclusion cette "distance" que nous avons définie ici, est une fonction propre à ces travaux, et ne vérifie pas les axiomes des distances utilisées en topologie.

d_{Max} (Distance maximale) : Soit $(u,v) \in V^2$, tels qu'il existe $n \in \mathbb{N}^*$ et $(v_0, \ldots v_n) \in \boldsymbol{AP_n}$, avec $v_0 = u$, et $v_n = v$. Alors $d_{Max}(u,v) = Max\{ n \in \mathbb{N}^* / \exists (v_0, \ldots v_n) \in \boldsymbol{AP_n},\ v_0 = u,\ and\ v_n = v \}$. Afin de s'affranchir des possibilités de boucles ou cycles, comme dans la figure 3-2 avec entre autres (a, g, e, a) ou (c, f, g, e, c), on utilise uniquement les chemins acycliques pour déterminer la distance maximale entre deux nœuds. Le contraire conduirait à considérer des distances qui seraient alors infinies. Dans la figure 3-2, $d_{max}(h, d) = 6$.

Ip (Entrée) : Soit un nœud $v \in V$, l'ensemble de ces entrées est défini par : $\boldsymbol{Ip}(v) = \{u \in V / (u, v) \in A\}$. Dans la figure 3-2, $Ip(g) = \{a, h, f\}$.

Pre (Prédécesseur) : Soit un nœud $v \in V$, l'ensemble de ces prédécesseurs est défini par: $\boldsymbol{Pre}(v) = \{u \in V / d_{min}(u, v) \in \mathbb{N}^*\}$. Dans la figure 3-2, $Pre(d) = \{a, b, c, e, f, g, h\}$, et $Pre(h) = \varnothing$. A noter également que les nœuds a, b, c, d, e, f et g ont tous le même ensemble de prédécesseurs.

Op (sortie) : soit un nœud $v \in V$, l'ensemble de ses sorties est défini par : $\boldsymbol{Op}(v) = \{u \in V / (v, u) \in A\}$. Dans la figure 3-2, $Op(g) = \{e\}$.

Suc (Successeur) : Soit un nœud $v \in V$, l'ensemble de ces successeurs est défini par : $\boldsymbol{Suc}(v) = \{u \in V / \boldsymbol{d_{max}}(v, u) \in \mathbb{N}^*\}$. Dans la figure 3-2, $Suc(d) = \varnothing$, et $Suc(h) = \{a, b, c, d, e, f, g\}$. A noter également que les nœuds a, b, c, e, f, g et h ont tous le même ensemble de successeurs.

3.2.2 Décomposition d'un graphe en plusieurs ensembles

Pour restaurer l'état des bascules contaminées par une erreur, la donnée corrigée fournie par le circuit de correction est utilisée. D'autres données provenant des états précédents d'autres bascules du circuit seront également utilisées. Ces états sont donc sauvegardés en ajoutant des FIFO à chacune de ces bascules, ainsi qu'un multiplexeur pour fournir les données des FIFO au circuit durant la décontamination. Durant les opérations normales les FIFO reçoivent à chaque cycle l'état précédent de la bascule. Donc une FIFO de q étages préservera les q derniers états de la bascule. Lors du premier cycle de décontamination, la FIFO fournira la donnée sauvegardée la plus ancienne, afin de ré-exécuter la $q^{ème}$ opération précédente. Puis, durant les cycles de décontamination, la FIFO est activée pour fournir, à chaque cycle, la valeur sauvegardée dans l'étage suivant de la FIFO. Ainsi on ré-exécute ensuite la $q–1^{ème}$ opération, puis la $q–2^{ème}$ etc.… Les bascules nécessitant une sauvegarde sont déterminées ci-après. Soit un circuit avec des mémoires protégées par un code correcteur avec une détection des erreurs faites en k cycles. Les bascules du système, modélisées par un graphe orienté, sont partitionnées en plusieurs ensembles. Pour illustrer ce partitionnement, nous nous appuierons sur le graphe de la figure 3-3.

LM (Lecture Mémoire) : L'ensemble LM contient tous les ports de lecture des mémoires du circuit, protégées par un code correcteur. Dans la figure 3-3, $LM=\{lm1, lm2\}$. Pour simplifier on admettra la convention suivante : $d_{min}(LM,v)=min\{d_{min}(lm,v), lm \in LM\}$. L'utilisation de cette convention est justifiée dans l'annexe A.

C_k (Nœuds contaminables en un nombre k ou moins de cycles d'horloge) : Cet ensemble est composé des bascules traversables par une donnée contaminée avant la détection de l'erreur. $C_k = \{ c \in V / d_{min}(LM, c) \leq k \}$. On notera aussi $C_k' = C_k \cup LM$. Dans la figure 3-3 $C_3=\{a, b, c, f, d, g, h, i\}$. On peut remarquer qu'il n'y a pas de différence entre les chemins qui proviennent de *lm1* et *lm2*. Cela permet de simplifier le problème, mais découle également du fait de l'utilisation de la convention précédemment citée.

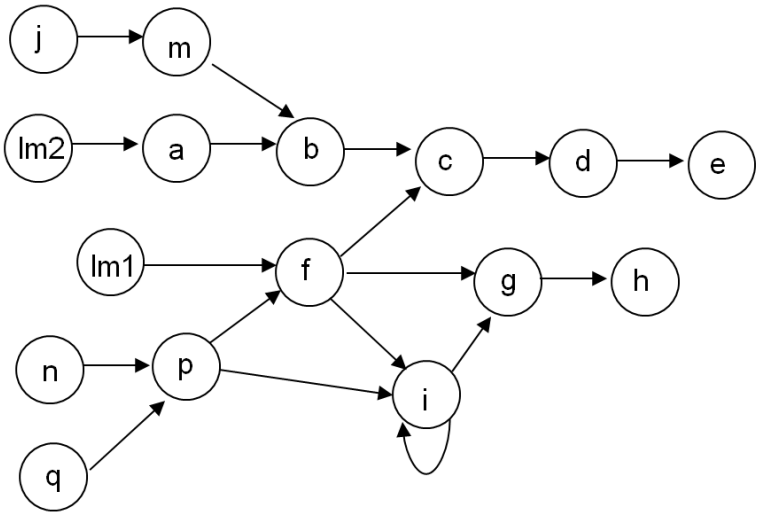

Fig. 3-3 : Exemple de digraphe pour illustrer les différents ensembles de nœuds

DC_k (Dernier contaminable) : Cet ensemble est composé des bascules contaminables uniquement au dernier cycle et pas avant. $DC_k=\{c\in C_k / d_{min}(LM, c)=k\}$. Ici $DC_3=\{d, h\}$.

NC_k (Non contaminable en k cycles) : Cet ensemble est composé des bascules ne pouvant pas être contaminées lors de la propagation d'une erreur. $NC_k = V \setminus C_k'$. Dans la figure 3-3 $NC_3=\{e, j, m, n, p, q\}$.

Une bascule dont l'état précédent sera utilisé pour la décontamination est appelée une source. Certaines sources appartiennent à l'ensemble C_k et d'autres à NC_k. L'utilisation d'un état précédent d'une source appartenant à C_k doit être faite avec prudence. En effet certains états sauvegardés dans la FIFO peuvent être contaminés, et ne doivent donc pas être utilisés. Cela peut arriver pour certains nœuds de C_k mais pas de DC_k. Un nœud de DC_k ne peut être contaminé qu'au $k^{ème}$ cycle. Donc l'état en question ne sera pas sauvegardé dans une FIFO, et sera décontaminé avant la réutilisation de ce nœud. A cause de cela on considérera de manière différente les nœuds de l'ensemble $C_k'\backslash DC_k$ les nœuds de l'ensemble $NC_k \cup DC_k$.

SE_k (Source extérieure) : SE_k est le sous-ensemble de $NC_k \cup DC_k$ dont les nœuds sont les prédécesseurs d'au moins un nœud de C_k. Comme ils précèdent un nœud contaminé, leurs précédents états pourront être requis pour procéder à la décontamination de ces nœuds contaminés. Dans la figure 3-3, $SE_3= \{j, m, n, p, q\}$.

$$SE_k=\{se\in NC_k \cup DC_k \ / \ C_k \cap Suc(se)\neq\emptyset\}$$

PMI_k (Prédécesseurs Multiple Immédiats) : L'ensemble PMI_k contient les nœuds de C_k qui ont plusieurs entrées appartenant à l'ensemble $C_k'\backslash DC_k$.

$$PMI_k = \{pmi\in C_k / Card(Ip(pmi) \cap C_k'\backslash DC_k) > 1\}$$

Dans la figure 3-3, $C_3'\backslash DC_3=\{a, b, c, f, g, i, lm1, lm2\}$, et $C_3=\{a, b, c, f, d, g, h, i\}$. Mais seuls les nœuds *c, f, g* et *i* de C_3 ont plusieurs entrées : $Ip(c)=\{b, f\}$; $Ip(f) =\{lm1, p\}$; $Ip(g) =\{f, i\}$; $Ip(i)=\{f, i, p\}$. Sur les 4 nœuds *c, f, g* et *i*, seul *f* ne vérifie pas la condition $Card(IP(f) \cap C_3'\backslash DC_3) >1$, car $p\notin C_3'\backslash DC_3$. Donc $PMI_3=\{c, g, i\}$.

SC_k (Source contaminable) : Soit $pmi\in PMI_k$ et l'ensemble associé $Pre(pmi)\cap C_k'\backslash DC_k$, suivant le circuit considéré, la donnée corrigée est susceptible d'atteindre *pmi*, avant que tous les nœuds de $Pre(pmi)\cap C_k'\backslash DC_k$ ne soient décontaminés. Ceux qui ne le sont pas encore sont des sources, car leurs états passés sont requis pour la décontamination de *pmi*, lorsque celui-ci reçoit la donnée corrigée provenant des autres chemins, plus courts. L'ensemble SC_k' contient ces sources qui sont plus éloignées de *LM* que le nœud *pmi* associé.

$$SC_k' = \bigcup_{pmi\in PMI_k} \{sc\in (Pre(pmi)\cap C_k'\backslash DC_k) / d_{min}(LM, sc) \geq d_{min}(LM, pmi)\}$$

Plus généralement, tout prédécesseur d'une de ces sources, et qui fait également partie de $C_k' \setminus DC_k$ peut servir à décontaminer *pmi.* En effet leurs états précédents peuvent être transmis aux nœuds de SC_k', et peuvent donc être considérés comme une source contaminable. Les prédécesseurs de ces sources qui ne font pas parties de $C_k' \setminus DC_k$ sont, par définition, des sources extérieures La réunion de ces deux ensembles, SC_k' et des prédécesseurs des éléments de SC_k' appartenant également à $C_k' \setminus DC_k$ constitue l'ensemble SC_k.

$$SC_k = \left(\bigcup_{sc \in SC_k} \left(Pre(sc) \cap C_k' \setminus DC_k \right) \right) \cup SC_k'$$

Dans la figure 3-3 rappelons que $PMI_3=\{c, g, i\}$. Par conséquent $SC_3'=\{b, i\}$. En effet $d_{min}(LM, b) \geq d_{min}(LM, c)$, et $d_{min}(LM, i) \geq d_{min}(LM, g)$ (ou alors de façon plus triviale $d_{min}(LM, i) \geq d_{min}(LM, i)$). $SC_3=\{lm1, lm2, a, b, f, i\}$.

SES_k (Source extérieure suffisante) : Un ensemble SES_k est un sous-ensemble de SE_k qui inclut un ensemble de nœuds dont les états précédents fournissent toutes les informations nécessaires pour restaurer l'état de C_k, exceptées les informations fournies par SC_k. Cet ensemble SES_k n'est pas unique.

SEI_k (Source extérieure immédiate) : SEI_k est le sous-ensemble de SE_k qui inclut chaque nœud *sei* de SE_k tel que *sei* ait au moins un successeur *c* appartenant à C_k.

$$SEI_k=\{sei \in ES_k / C_k \cap Op(ies) \neq \emptyset\}.$$

Par construction, SEI_k est l'un des SES_k possibles, car : $\forall c \in C_k\ Ip(c) \subset SEI_k \cup C_k'$. Dans la figure 3-3 $SEI_3=\{m, p\}$.

SCS_k (Source contaminable suffisante) : Un ensemble SCS_k est un sous-ensemble de SC_k incluant un ensemble de sources permettant de fournir toutes les informations de SC_k requises pour la décontamination de l'ensemble PMI_k, exceptées les informations fournies par l'ensemble SE_k. Cet ensemble SCS_k n'est pas unique.

SCI_k (Source contaminable immédiate) : Le sous-ensemble de SC_k qui inclut chaque nœud *sci* de SC_k tels que *sci* ait, au moins, une sortie $pmi \in PMI_k$ décontaminée avant *sci*.

$$SCI_k = \{sci \in SC_k / \exists pmi \in PMI_k \cap Op(sci) / d_{min}(LM, sci) \geq d_{min}(LM, pmi)\}$$

Par construction, SCI_k est l'un des SCS_k possibles. En effet, soit un nœud $pmi \in PMI_k$ et une entrée $v \in Ip(pmi)$, $v \notin SE_k \cup SCI_k$. Alors $d_{min}(LM, v) < d_{min}(LM, pmi)$. Donc *v* sera décontaminée avant pmi. Dans la figure 3-3 $SCI_3=\{b, i\}$. On vérifie ainsi, que ce soit pour SEI_k ou SCI_k, qu'un nœud *c* n'a potentiellement besoin d'une source immédiate que si $Card(Ip(c))>1$.

Les ensembles SES_k et SCS_k n'étant pas uniques, on peut créer un ensemble d'ensembles de type SES_k, et un ensemble d'ensembles de type SCS_k. Un ensemble de sources permettant la décontamination du circuit est constitué par la réunion de deux ensembles, l'un de type SES_k, l'autre de type SCS_k. Les FIFO (ainsi que leurs cycles d'activation) associées, permettant d'effectuer cette décontamination sont décrites dans la section suivante.

3.2.3 FIFO associée aux nœuds sources, et cycles d'activation

Pour décontaminer les états des bascules de l'ensemble C_k, on doit fournir à chaque nœud d'un des ensembles $SES_k \cup SCS_k$ une FIFO et un multiplexeur, comme expliqué dans le chapitre précédent. La taille des FIFO et les cycles durant lesquels FIFO et MUX sont activés sont détaillés en dessous. Le cas de l'ensemble $SEI_k \cup SCI_k$ est considéré en premier, car il induit en général le minimum de matériel supplémentaire. C'est également la solution telle que présentée dans la section 2.5.

FIFO pour les nœuds de SEI_k : Soit un nœud *sei* et un successeur immédiat *c* appartenant à C_k. La taille de la FIFO devant être utilisée pour *sei* doit être égale à $k–d_{min}(LM, c)+1$. La FIFO commence à transmettre ses données au $d_{min}(LM, c)$ cycle de la décontamination (cycle durant lequel *c* est atteint par la donnée corrigée provenant de la mémoire), et arrête de fournir des données au k^{th} cycle du processus (les derniers nœuds ont été contaminés). Dans certains cas, *sei* peut avoir plusieurs successeurs immédiats appartenant à l'ensemble C_k. Ce qui a été dit précédemment est toujours valable si on remplace $d_{min}(LM, c)$ par *:* $min\{d_{min}(LM, c), c \in C_k \cap Op(sei)\}$. Autrement dit, la FIFO est choisie aussi grosse que nécessaire, et commence à transmettre ses données dès que besoin. Dans la figure 3-3, et toujours avec $k=3$, le nœud *m* est associé à une FIFO de deux étages. Les données commencent à être émises lors du deuxième cycle de décontamination jusqu'au troisième. Le nœud *p* est associé à une FIFO de taille 3, laquelle est active durant tous les cycles de décontamination.

FIFOs des noeuds SCI_k : Nous avons aussi besoin de sauvegarder l'état précédent des nœuds de SCI_k avec une FIFO et un MUX pour transmettre les données au circuit. On procède de la même manière que pour SEI_k. Cependant durant la décontamination, on doit arrêter de transmettre les données de la FIFO pour éviter de réémettre des données contaminées. Soit *pmi* un nœud de PMI_k, et *sci* un prédécesseur immédiat de *pmi* appartenant à SCI_k. La taille de la FIFO utilisée pour sauvegarder les états de *sci* est de $k – d_{min}(LM, pmi)+ 1$. La FIFO commence à fournir des données au $d_{min}(LM, pmi)$ cycle de décontamination, et s'arrête au $d_{min}(LM, sci)+ 1$ cycle de décontamination (*sci* a été décontaminé). Après quoi, le MUX transmet de nouveau les données de *sci*. Au début de la décontamination, la première donnée contaminée, à l'intérieur de la FIFO se trouve stockée à l'étage $k – d_{min}(LM, sci)$, en comptant depuis l'entrée de la FIFO. Par conséquent il reste $d_{min}(LM, sci) – d_{min}(LM, pmi) + 1$ étages qui ne sont pas contaminés. Au vu des cycles (décrits précédemment) pendant lesquels la FIFO est activée, toutes ces données sont utilisées pendant la décontamination. Lorsque

le MUX utilisera à nouveau les données de *sci*, la première donnée contaminée sera stockée dans le dernier étage de la FIFO. Par conséquent, les données erronées ne sont pas réintroduites dans le circuit durant la décontamination. De la même manière que pour le cas précédent, si *ics* a plusieurs successeurs immédiats appartenant à PMI_k, $d_{min}(LM, pmi)$ est remplacé par : $min\{d_{min}(LM, pmi), pmi \in PMI_k / d_{min}(csi, pmi)=1\}$

Dans la figure 3-3 le nœud *i* est associé à une FIFO de taille 2 mais qui n'est active que durant le deuxième cycle de décontamination. Le nœud *b* constitue un exemple de cas de figure où l'ensemble $SEI_k \cup SCI_k$ ne fournit pas une solution optimisée au niveau hardware. En effet alors que *m* est un nœud source, et que la mémoire *lm 2* peut (grâce au pipeline du bloc de correction) retransmettre les données précédentes, *b* est ici considéré comme un nœud source. Une FIFO de taille 2 lui est associé. Comme pour le nœud *i* celle-ci est active durant le second cycle de décontamination.

FIFO pour les nœuds de $SES_k \cup SCS_k \neq SEI_k \cup SCI_k$: L'utilisation de sources extérieures *ses*, qui ne sont pas des prédécesseurs immédiats d'un nœud de *C*, implique qu'atteindre le premier nœud *c* de C_k prendra $d_{Max}(ses, c)$ cycles. On a besoin de $d_{Max}(ses, c)$ cycles car tous les chemins acycliques doivent être traversés pour une décontamination correcte. En effet toutes les entrées de *c* doivent être dans un état correct pour décontaminer *c*. Donc tous les nœuds entre *ses* et *c* doivent être décontaminés, et donc pour effectuer cette décontamination d'autres sources sont éventuellement placées en rapport. Cependant il n'y a pas besoin de restaurer un état fourni par un autre élément de SES_k. Dans ce cas particulier pour le calcul de $d_{Max}(ses, c)$, les chemins $(v_0, \ldots v_n) \in \boldsymbol{AP_n}$, $v_0=ses$, $v_n=c$ tels que $\forall i, 0 < i < n, v_i \notin SES_k$ sont considérés. De plus les nœuds de ces chemins doivent être débloqués pendant le processus de décontamination pour acheminer l'état correct jusqu'au nœud *c*. La taille de la FIFO pour restaurer l'état de *c* est $k - d_{min}(LM, c) + d_{Max}(ses, c)$. Si le nœud *c* n'est pas unique, la taille de la FIFO sera déterminée par rapport au cas le plus défavorable pour tous les nœuds *c* concernés.

Si la taille d'une FIFO est plus large que *k*, alors le processus de décontamination excède *k* par un nombre de cycles égal à la plus grande des *FIFO – k*. La FIFO associée au nœud $ses \in SES_k \setminus IES_k$ sera activée au $d_{min}(LM, c) - d_{Max}(ses, c) + 1$ cycle d'horloge. Les valeurs négatives impliquent que cela se produit avant que les données corrigées ne soient prêtes (ou transmises si le nombre de cycles en plus est trop grand).

De façon équivalente, pour les sources contaminées on considère un nœud *scs* de SCS_k et le plus éloigné des nœuds *mip* appartenant au MIP_k associés, atteint sans traverser un autre nœud de SCS_k ou MIP_k. Le nœud *scs* a besoin d'une FIFO de taille $k - d_{min}(LM, mip) + d_{Max}(scs, mip)$. Comme dans le cas précédent elle est active plus tôt, que celles des nœuds de ICS, au cycle $d_{min}(LM, mip) - d_{Max}(scs, mip) + 1$.

Dans l'exemple de la figure 3-3 la source du nœud *b* peut être remplacée par une FIFO associée au nœud *m* de taille 3 (au lieu de 2). De plus on ajoute un étage de pipeline au bloc de correction de *lm 2*, et ainsi on peut réémettre les 4 dernières données lues en mémoire (au lieu des 3 dernières seulement). Le nœud *a* est débloqué au cycle 0, et le nœud

b (ainsi que le nœud *f*) au cycle 1. Le nœud *c* peut alors être décontaminé au cycle 2. Dans cet exemple, on peut noter qu'une source contaminable (*b*) a été remplacée par deux autres sources, dont une non contaminable (*m*). Ceci montre que, si les ensembles SEI_k et SCI_k peuvent être déterminés séparément, les ensembles SES_k et SCS_k sont généralement interdépendants. On ne doit pas les considérer séparément, mais leur réunion $SES_k \cup SCS_k$, ainsi que leurs FIFO associées, pour savoir s'il n'y a pas des redondances inutiles que l'on peut supprimer, et ainsi réduire le coût matériel.

IMPORTANT : Quel que soit le cas considéré ($SES_k \cup SCS_k$ ou $SEI_k \cup SCI_k$) la taille d'une FIFO associée à un nœud *s* (*FIFO(s)*), et le premier cycle d'activation de la FIFO (*clk(s)*) sont liés par la relation : $clk(s) = k - FIFO(s) + 1$. La taille de la FIFO et les cycles d'activations sont dimensionnés pour atteindre le nœud contaminable le plus éloigné depuis la source *s*. Les sources sont positionnées de manière à fournir toutes les données non fournies par la mémoire. Ce qui assure une décontamination correcte.

3.2.4 La question de l'optimisation

La solution $SEI_k \cup SCI_k$ permet d'avoir une solution optimisée du point de vue du temps de décontamination. En effet les ressources étant placées au plus près des nœuds à décontaminer, elles sont immédiatement disponibles. Lorsqu'on utilise une solution différente, la discussion précédente montre que des cycles d'horloge supplémentaires peuvent être nécessaires. En revanche, cela peut conduire à une solution qui minimise le coût du matériel supplémentaire. La question de "comment caractériser une solution $SES_k \cup SCS_k$?" se pose alors. En effet si tous ces ensembles sont connus, il suffit de choisir celui qui minimise la somme de toutes les FIFO associées. Une condition nécessaire toutefois pour minimiser le coût matériel est de supprimer les redondances inutiles comme dans l'exemple du paragraphe précédent.

Pour qu'un ensemble soit déclaré solution, celui-ci doit permettre de fournir les mêmes informations que l'ensemble $SEI_k \cup SCI_k$. Par exemple, dans la figure 3-3 et pour $k=3$, la FIFO associée au nœud *p* peut-être supprimée, à condition de pouvoir fournir les données qu'aurait sauvegardé cette FIFO d'une autre façon. Il faut donc que les données de *n* et *q* soient valides au cycle précédent. Cependant, cela implique que chacun des nœuds soit associé à une FIFO ayant un étage de plus que celle associée à *p*. Toutefois, si on ne considère que cette partie du graphe formée par ces trois nœuds *(n, p* et *q)*, il y a déjà cinq ensembles qui sont solutions : *{p}, {n, q}, {n, p}, {p, q}* et *{p, n, q}*. Dans les trois derniers ensembles il y a des redondances inutiles, mais rien n'interdit de les considérer comme des ensembles solutions.

Pour éviter la redondance on peut remplacer un nœud *s*, dans un ensemble solution *S*, par ses entrées : $S' = S \setminus \{s\} \cup Ip(s)$. S' est alors un nouvel ensemble solution. La contrepartie étant une FIFO associée aux éléments de *Ip(s)* ayant un étage de plus que les éléments de *s*. Dans le cas le plus général cela ne réduit pas le coût matériel, sauf si par exemple $Ip(s) \subset S \setminus \{s\}$, ce qui revient à supprimer la redondance inutile. L'ensemble $SEI_k \cup SCI_k$ est

constitué de nœuds se trouvant au plus près des nœuds à décontaminer. Par conséquent un nœud d'un ensemble $SES_k \cup SCS_k$ quelconque est soit un nœud de l'ensemble $SEI_k \cup SCI_k$, soit un prédécesseur d'un nœud de ce même ensemble. Par conséquent, en commençant à partir de l'ensemble $SEI_k \cup SCI_k$, et en appliquant le processus de remplacement décrit ci-dessus pour chaque élément de chaque nouvel ensemble de solution trouvé, on obtient alors la solution qui minimise le coût matériel. L'inconvénient de ce processus itératif est le nombre de solution à examiner, qui peut devenir énorme sur un exemple réel. En plus de ce processus, il faut également ajouter une fonction qui évalue le coût de chaque solution, afin de déterminer laquelle est la moins coûteuse. Dans la partie algorithmique nous présenterons une méthode pour obtenir une solution sous-optimale, mais qui a permis d'avoir des résultats significatifs en des temps raisonnables.

3.3 Les étapes de l'algorithme

Les éléments de formalisme que nous venons de présenter, permettent d'établir des algorithmes pour automatiser l'implémentation de la solution. Les différentes étapes sont illustrées par le schéma de la figure 3-4. D'abord nous évoquerons la manière dont le circuit est décrit, puis comment on obtient l'ensemble des nœuds contaminés décrits dans la section 3.2. Une fois que ceux-ci sont connus, la taille de la FIFO associée aux nœuds sources dans les ensembles $SEI_k \cup SCI_k$ s'obtient facilement. Nous expliquerons ensuite les suppositions et la méthode employée pour optimiser la solution obtenue du point de vue du matériel additionnel. Il convient enfin d'établir les différentes étapes de la décontamination nécessaires à l'établissement de la FSM contrôlant la décontamination. Les entrées et sorties du circuit sont décrites dans plusieurs fichiers textes.

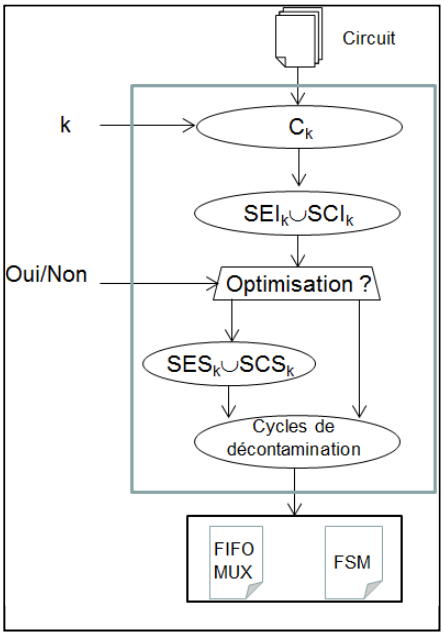

Fig. 3-4 : Les étapes de l'algorithme

On peut donc diviser l'outil en cinq parties :

- La lecture des fichiers décrivant le circuit ;
- L'identification des bascules contaminées C_k ;
- L'identification des sources extérieures immédiates pour la décontamination (SEI_k), et celles atteignables (SCI_k) ainsi que leurs FIFO associées ;
- L'optimisation matérielle ;
- La description des phases de décontamination.

3.3.1 Description d'un circuit

Les fichiers décrivant un circuit sont divisés en deux parties : définition des composants utilisés et leurs interconnections. Un composant est défini par :

- un type (mémoire, registre, FIFO, logique),
- un label identifiant les composants,
- le nombre de bits stockés pour les composants de stockage,
- un paramètre optionnel (taille des FIFO…).

Dans le cas des circuits complexes, une description hiérarchique (comme en VHDL ou en Verilog) est plus adaptée. Pour cela il est possible de créer de nouveaux types de composants dans d'autres fichiers, et de les utiliser de la même manière à condition de préciser en plus les ports d'entrées et de sorties. La description hiérarchique est illustrée par la figure 3-5. A noter que les composants *U1* et *U2* peuvent avoir le même type (tel que dans la figure), ou avoir des types différents. Les ports d'entrées/sorties sont précisés par les lettres e/s dans la figure 3-5. Le nombre de ports ou de niveau de hiérarchisation n'est théoriquement pas limité (en pratique il est limité par les capacités de l'outil de simulation), et il est donc possible ainsi de créer une véritable bibliothèque de composant.

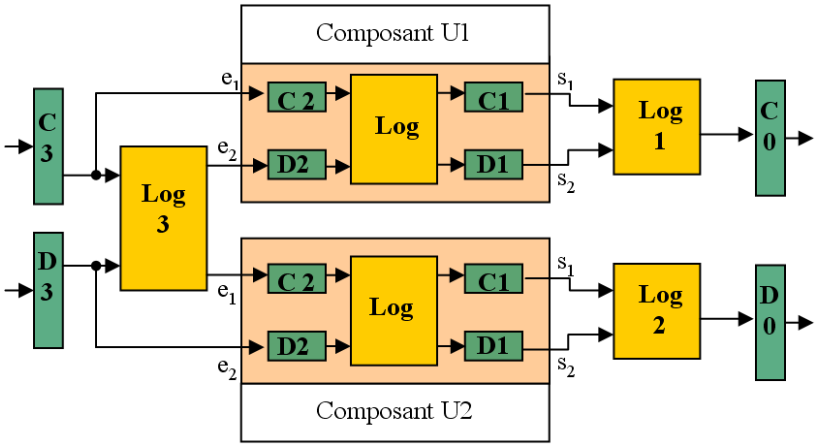

Fig. 3-5 : Description hierarchique d'un circuit

Pour être plus précis dans la description du circuit on peut introduire la capacité de préciser différentes configurations du circuit en ajoutant dans la bibliothèque des composants un composant de type multiplexeur. De manière évidente, le multiplexeur doit avoir la même configuration durant les phases de contamination et les phases de décontamination. Cela peut conduire à des possibilités permettant de réduire le coût matériel de notre solution, c'est à dire le nombre et la taille des FIFO à ajouter. Deux types de configuration peuvent être utilisés dans la description du circuit : statique ou dynamique. Une configuration statique sera utilisée pour des circuits ayant une architecture reconfigurable pour effectuer plusieurs algorithmes, mais dont la configuration ne change pas pendant l'exécution d'un algorithme. Autrement dit, on peut affirmer que lors des cycles de contamination et des cycles de décontamination du circuit la configuration ne change pas. Cela permet de réduire le coût matériel, comme dans l'exemple suivant.

Supposons, dans la figure 3-6, que le *module 1* et le *module 2* soient composés d'un nombre d'étages de pipeline différents (ex. 1 pour le *module 1* et 3 pour le *module 2*). Supposons également que le nombre de cycles avant le blocage du circuit soit de 3. Si on ne tient pas compte du multiplexeur, le dernier étage du *module 2* doit comporter une FIFO additionnelle de deux étages. En revanche si on tient compte du multiplexeur, et si on suppose que l'on est dans le cas d'une configuration statique, une FIFO additionnelle n'est pas nécessaire. En effet, durant les cycles de décontamination ne seront utilisées que les données provenant d'un seul des deux modules. Ceci n'est plus valable dès que la configuration du multiplexeur est susceptible de changer durant la phase de décontamination. En effet, durant le premier cycle le *module 1* est décontaminé. Au deuxième le multiplexeur transmet la donnée correcte provenant du *module 1* au reste du circuit (dans le cas contraire il n'y aurait pas de décontamination du reste du circuit nécessaire). Supposons alors que la configuration du *MUX* change au cycle suivant. Le *module 2* comportant 3 étages, ce dernier n'est pas encore décontaminé au moment où une donnée correcte est requise. Une FIFO, d'un seul étage, est donc nécessaire, alors que sans tenir compte du *MUX* il faudrait une FIFO de deux étages.

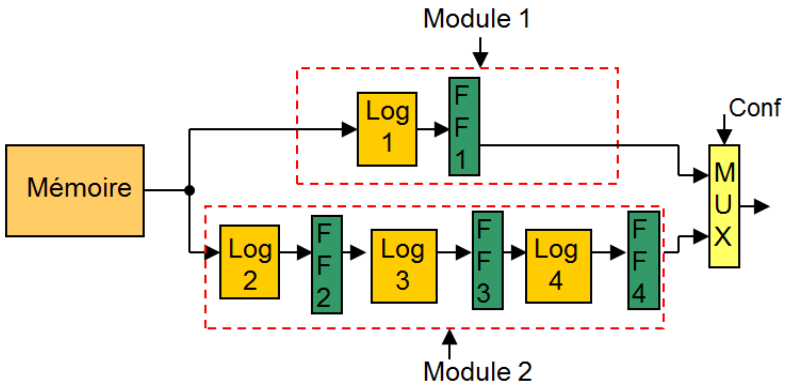

Fig. 3-6 : Utilisation d'un MUX à la convergence de chemins de longeurs différentes

Une configuration dynamique constitue le cas le plus général dans un circuit comportant des multiplexeurs, où chacun d'entre eux est susceptible de changer de configuration à chaque cycle d'horloge. Dans le cas le plus général, tenir compte de ces multiplexeurs ne permet pas de réduire le coût matériel. Toutefois, il y a des cas où la taille des FIFO rajoutées peut être tout de même réduite, comme dans l'exemple précédent. Cependant, il faut étudier ces cas avec prudence. La démonstration en est faite avec les deux figures suivantes. Dans la figure 3-7 (a) supposons dans un premier temps que l'on ne tienne pas compte du multiplexeur. On doit associer au registre *REG2* une FIFO avec un nombre d'étages égal au nombre de cycles d'horloge avant la détection de l'erreur. Si le circuit est bloqué en un coup d'horloge, il faut associer à *REG2* une FIFO ayant un étage. Supposons maintenant que l'on tienne compte du multiplexeur. La donnée erronée n'est propagée dans le circuit que si le multiplexeur est configuré pour transmettre, durant le cycle où la donnée erronée est lue, la donnée provenant de la mémoire. Lorsque la donnée corrigée est émise, la donnée provenant de *REG2* n'est pas utilisée (car le multiplexeur doit être configuré durant les cycles de décontamination de la même manière que durant les cycles où l'erreur est propagée). La conséquence est qu'il n'y a pas besoin de rajouter une FIFO associée à *REG2*. De manière plus générale, si la donnée est propagée pendant *k* cycles, une FIFO de *k-1* étages associée à *REG2* est suffisante. A l'inverse la figure 3-7 (b) démontre que ce genre de considération n'est plus valable pour un circuit à peine différent. En effet ici il y a deux multiplexeurs, dont les configurations sont inversées. Par conséquent si l'un des deux MUX transmet la donnée provenant de la mémoire, l'autre transmet la donnée provenant de *REG2*. Les deux données sont ensuite utilisées dans la même fonction logique *Log1*, avant que le résultat ne soit stocké dans *REG1*. Il n'y a donc pas de réduction de matériel possible, car la donnée provenant de *REG2* doit être réémise durant le cycle de décontamination, et la taille de la FIFO qui doit être associée à *REG2* est identique au cas où l'on ne considère pas de multiplexeur.

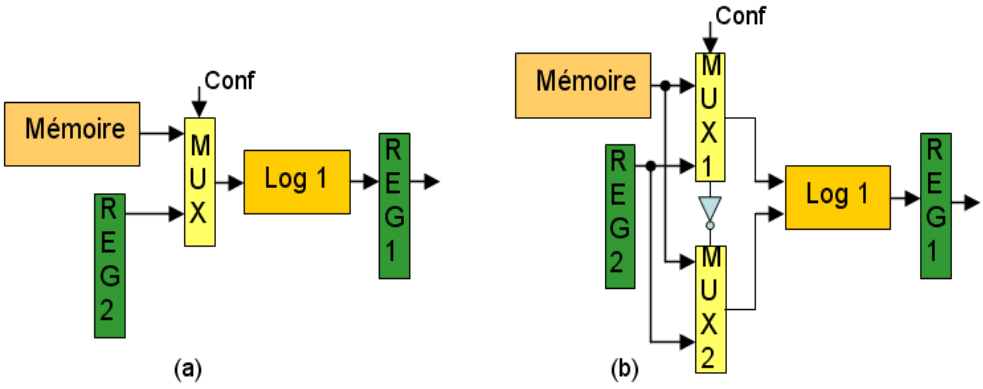

Fig. 3-7 : Utilisation d'un MUX dans le cas de sources extérieures

Ce problème pourrait théoriquement être résolu grâce à l'algèbre de Boole. Dans le cas de la figure 3-7 (a), on pourrait écrire $R1 = M \cdot Cf + R2 \cdot \overline{Cf}$, où Cf et $\overline{Cf}$ représentent les deux configurations possible du multiplexeur. A l'inverse pour la figure

3-7 (b), les équations seraient les suivantes : $Mux1 = M \cdot Cf + R2 \cdot \overline{Cf}$ et $Mux2 = M \cdot \overline{Cf} + R2 \cdot Cf$, avec $R1 = Mux1 + Mux2 = M + R2$. On peut donc traduire ainsi si les multiplexeurs ont ou non un impact sur la propagation des données dans le circuit. Cette méthode est pourtant incomplète. En effet reconsidérons une nouvelle fois la figure 3-6. Si on ne tient pas compte de l'éventuelle différence dans le nombre d'étages entre les deux modules, on a l'équation $Mux = M \cdot Cf + M \cdot \overline{Cf}$. Autrement dit les équations doivent être établis en tenant compte des cycles d'horloge.

De plus, dans la section 2.6, et 3.2.3 de ce chapitre, on a parlé de la possibilité de commencer la décontamination avant que la donnée corrigée soit prête. Ce dernier cas de figure annule les avantages dus à la prise en compte des multiplexeurs. En effet, reconsidérons à nouveau la figure 3-7 (a). Si la décontamination se fait en *k* cycles, une FIFO de *k-1* étages est suffisante, comme précisé auparavant. En revanche, si la décontamination commence avant que la donnée corrigée ne soit prête, les données précédentes peuvent soit venir de la mémoire, soit de *REG2*, ce qui n'autorise aucune optimisation de la taille de la FIFO associée à *REG2*.

En conclusion, lorsqu'on souhaite savoir si en tenant compte des multiplexeurs on peut réduire le coup matériel, il convient de procéder à une étude préliminaire qui peut être complexe. Cette démarche, provoque une augmentation de la durée de simulation de manière significative. Ceci est d'autant plus vrai que dans le cas le plus général, si toutes les configurations sont permises durant les *k* cycles où l'erreur se propage, il n'y a pas d'optimisation possible. Par conséquent sauf dans des cas particuliers, nécessitant une étude au préalable du designer, on évitera de tenir compte des multiplexeurs.

3.3.2 Identification des chemins contaminables

Lorsque le circuit est décrit, il faut préciser le nombre de cycles d'horloge nécessaire à la détection d'une erreur, en fonction du code correcteur que l'on utilise. L'outil identifie alors tous les chemins partant des ports de sortie des mémoires et ayant *k* cycles d'horloge de longueur (CP_k). De cette façon on obtient tous les chemins contaminables. Cette étape est illustrée par les figures 3-8 (a) et (b), ainsi que le tableau 3-1. La figure 3-8 (a) représente une partie d'un circuit avec le niveau d'abstraction utilisé par notre algorithme. La figure 3-8 (b) représente le même circuit sous la forme d'un graphe orienté. Dans cet exemple si *k=1*, le seul chemin contaminable est CP_1*={(LM, N3)}*. Si *k=2*, il y a deux chemins contaminables : CP_2*={(LM, N3, N2), (LM, N3, N4)}*. Si *k=3*, il y a quatre chemins contaminables : CP_3*={(LM, N3, N2, N2), (LM, N3, N2, N1), (LM, N3, N2, N4), (LM, N3, N4, N0)}*. En même temps que l'ensemble des chemins contaminés, on a du même coup l'ensemble C_k des composants contaminés : C_1*={N3}* ; C_2*={N3, N2}* ; C_3*={N3, N2, N1, N4, N0}*. Tous les chemins sont sauvegardés pour pouvoir identifier ensuite les ensembles de sources immédiates. Pour identifier ces ensembles, pour chaque chemin et pour chaque nœud, le cycle d'horloge auquel le nœud est atteint est également sauvegardé. Le résultat concernant la figure 3-8, et pour *k=3* est

résumé dans le tableau 3-1.

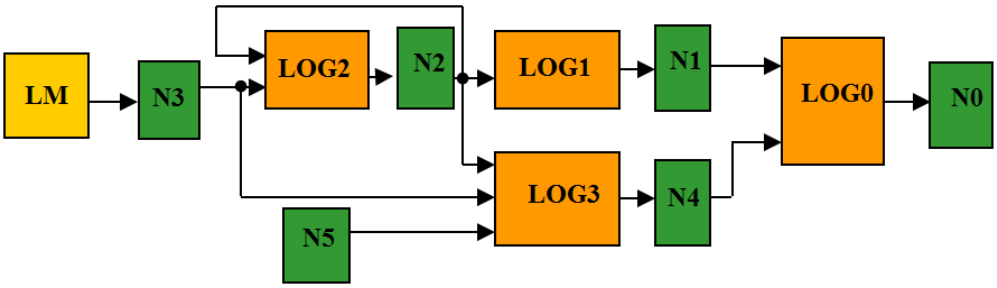

Fig. 3-8 (a) : Exemple de circuit pour illustrer les étapes de l'algorithme

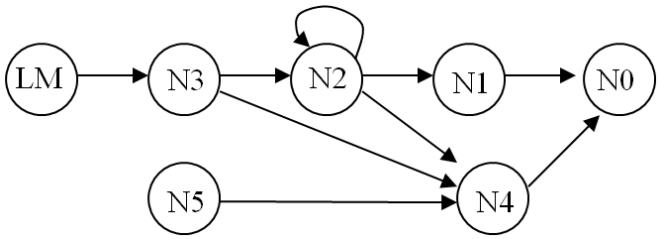

Fig. 3-8 (b) : Représentation sous forme de digraphe de la Fig. 3-8 (a)

Tab 3-1: Les chemins contaminés pour k=3 de la figure 3-5 (b)

Clk	CP_3 1	CP_3 2	CP_3 3	CP_3 4
0	LM	LM	LM	LM
1	N3	N3	N3	N3
2	N2	N2	N2	N4
3	N1	N2	N4	N0

Cette première étape dont on vient de parler est effectuée par un algorithme de parcours en profondeur adapté à notre problème. Un parcours en profondeur explore les chemins d'un graphe un par un. A chaque étape, l'algorithme explore le premier nœud connecté en sortie du nœud actuel, et ainsi de suite, de proche en proche, jusqu'à atteindre un nœud n'ayant plus de nœuds connectés en sortie (ou que toutes ses sorties aient été explorées). L'algorithme (Annexe B) revient alors au dernier nœud exploré dont tous les nœuds connectés en sortie n'ont pas été visités. Ici la "profondeur" de l'exploration est limitée par le nombre *k*. Donc, si on considère le circuit représenté sous forme de graphe orienté, on considère tous les chemins de longueur *k* partant des nœuds représentant les ports de sortie des mémoires. On limite donc le nombre de nœuds traversés, contrairement à une exploration en profondeur classique. En revanche, afin d'éviter que l'exploration ne prenne un temps infini, un parcours en profondeur classique n'explore jamais deux fois le même nœud pour un même chemin. En effet, si le graphe comportait des boucles, cela conduirait à les parcourir de manière illimitée. Ici, puisque la longueur des chemins est limitée, cette restriction ne s'applique pas, et chacun des chemins obtenus peut contenir plusieurs fois le même nœud.

3.3.3 Les ensembles de sources immédiates SEI et SCI

Une fois tous les chemins contaminés connus, les ressources dont on a besoin pour la décontamination de ces chemins peuvent être identifiées, à commencer par les sources immédiates. Pour cela, les entrées de chaque nœud du graphe appartenant à ces chemins sont vérifiées, en particulier les nœuds qui ont plusieurs entrées. En effet, les sources extérieures ou contaminables ne peuvent exister que si un nœud contaminé possède plusieurs entrées (voir Section 3.2.2). Lors de la décontamination, l'état valide d'un nœud ne peut être restauré que si toutes ses entrées sont valides. Si certaines de ces entrées n'appartiennent pas au chemin contaminé, alors ce sont des sources extérieures. Si plusieurs entrées d'un nœud appartiennent à un chemin contaminé, leurs distances minimales depuis les sorties des mémoires doivent être comparées (Section 3.2.2). Cette distance est facilement obtenue à partir des chemins déterminés à l'étape précédente. Si $cp=(cp(1), cp(2)\ldots cp(k+1))$ est un chemin de CP_k et $cp(i)$ le $i^{ème}$ nœud de ce chemin, la distance $d_{min}(LM, c)$ définie dans la section 3.2.1 est déterminée par:

$$d_{min}(LM,c)=min\{i \in \mathbb{N}^* / cp(i)=c,\ cp \in CP_k\} - 1.$$

Si une entrée de c est connectée à un nœud c' qui n'est pas listé dans un de ces chemins, la donnée fournie par ce nœud provient d'une source extérieure (ensemble SE_k). Il est identifié comme étant un nœud de SEI_k, avec une FIFO de $k–d_{min}(LM, c)+1$ étages. Si ce nœud a déjà été identifié comme appartenant à SEI_k, la taille de la FIFO est mise à jour en gardant la plus grande des deux. Pour la figure 3-8 et $k=3$, le nœud *N5* est un nœud de SEI_k avec une FIFO de taille 2. Si l'entrée d'un nœud c est connectée à plusieurs nœuds listés dans un des chemins identifiés, les cycles d'horloge associés de ces nœuds sont comparés entre eux. Si un ou plus sont plus petits que les autres, les nœuds qui n'ont pas la plus petite des valeurs sont identifiés comme des nœuds de SCI_k (ou SEI_k si le composant est contaminé seulement durant le dernier cycle), avec une FIFO associée de $k– d_{min}(LM,c)+1$ étage, ou la plus grande si elle était déjà identifiée comme étant un nœud de SCI_k. Dans la figure 3-8, avec $k=3$, *N1* et *N2* sont respectivement des nœuds de SEI_k et SCI_k avec une FIFO de 1 et 2 étages. Ces FIFO sont ajoutées à celle associée à *C5* dont on a parlé précédemment. L'algorithme détaillé de cette étape est donné dans l'annexe C.

A la fin de cette étape, l'ensemble $SEI_k \cup SCI_k$ et leur FIFO associée sont une solution pour implémenter la solution décrite dans la section 3.2. Cette solution est optimisée du point de vue du nombre de cycles d'horloge, puisque la décontamination n'a pas à démarrer avant que la donnée corrigée ne soit fournie par le circuit de correction. Cependant elle peut être non optimisée du point de vue du matériel.

3.3.4 Optimisation matérielle

Soit un ensemble de sources $SES_k \cup SCS_k$ (qui ne sont pas déterminés à ce stade de l'algorithme), on considère qu'il y a trois paramètres qui permettent d'évaluer le coût matériel de cette solution :

1– le nombre de sources (où le cardinal de l'ensemble $SES_k \cup SCS_k$) ;

2– la taille d'une FIFO associée à une source $s \in SES_k \cup SCS_k$ (notée *FIFO(s)*) ;

3– le nombre de bits stockés dans la source s (noté *SIZE(s)*).

De façon triviale, on peut dire que chacun des *FIFO(s)* étages de la FIFO associée à s stocke *SIZE(s)* bits de données. On assume le fait que le coût d'une FIFO est proportionnelle au nombre d'étage et au nombre de bits stockés par étage. Pour simplifier la notation on notera S un ensemble de sources $SES_k \cup SCS_k$ quelconque. A cet ensemble S on y associe la fonction coût J suivante :

$$J(S) = \sum_{s \in S} SIZE(s) \times FIFO(s)$$

Cette fonction coût évalue le nombre total de bits de données stockées. On assumera le fait que minimiser le coût matériel revient à trouver l'ensemble solution qui minimisera cette fonction coût. Dans le cas où deux ensembles S_1 et S_2 minimisent la fonction J, on gardera la solution dont le cardinal est minimum, c'est-à-dire qui comporte le moins de sources. De cette manière on minimise le nombre de multiplexeurs additionnels. On notera dans la suite que l'algorithme de recherche de solution est indépendant de la fonction coût. Par conséquent rien n'interdit d'utiliser une fonction coût qui serait éventuellement plus réaliste, les fonctions étant alors interchangeables.

L'étape précédente a identifié l'ensemble $SEI_k \cup SCI_k$. Ces sources sont situées au plus prêt des nœuds qu'elles sont sensées décontaminer. Par conséquent toute source est soit un élément de $SEI_k \cup SCI_k$, soit un prédécesseur d'un élément de $SEI_k \cup SCI_k$. Tous les ensembles S peuvent être trouvés à partir de $SEI_k \cup SCI_k$. Un nouvel ensemble S' est trouvé en remplaçant un nœud s d'un ensemble S par les entrées de ce nœud *(Ip(s))*. Toutefois comme les nouvelles sources sont plus éloignées des nœuds à décontaminer que la précédente, il faut leur associer à chacun une FIFO ayant un étage de plus que celle associée à s. Si toutefois l'un des éléments de *Ip(s)* était déjà associé à une FIFO, on lui attribue la FIFO ayant le plus grand nombre d'étages.

$$\begin{cases} S' = Ip(s) \cup S \setminus \{s\} \\ \forall s' \in Ip(s), FIFO(s') = Max(FIFO(s'), FIFO(s) + 1) \end{cases}$$

Dans la plupart des cas, le coût matériel de la solution S' est plus important que celle de la solution S. En appliquant ce principe, la taille d'une FIFO étant soit identique soit plus grande, pour réduire le coût matériel J on peut soit jouer sur la taille des données stockées (paramètre *SIZE(s)*), soit sur le nombre de sources (paramètre *Card(S)*). La première option est illustrée par la figure 3-9. Dans la partie supérieure (a) une FIFO de taille L est attribuée à *REG2*. Dans la partie supérieure (b) on a choisi de déplacer cette FIFO en l'associant à *REG1*. La FIFO possède alors $L+1$ étages. Cette solution réduit le coût matériel si et seulement si :

$$SIZE(REG1) < \frac{L}{L+1} \times SIZE(REG2)$$

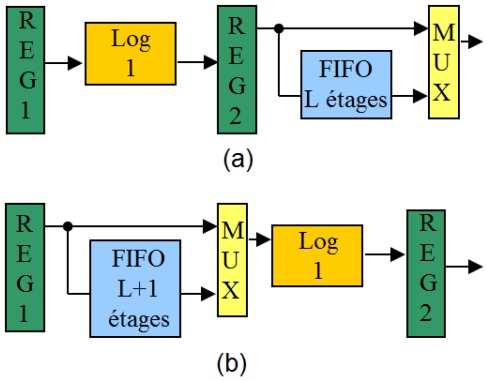

Fig. 3-9 : Optimisation du coût matériel en jouant sur la taille du registre

La deuxième option est illustrée par la figure 3-10. Les deux cas représentent les mêmes parties d'un même circuit avec différents ensembles $SES_k \cup SCS_k$. Dans le cas (a), deux nœuds *C1* et *C2* sont protégés par une FIFO de *L* étages chacune. Dans le cas (b), les deux FIFO de *L* étages associées à *C1* et *C2* ont été remplacées par une FIFO de *L+1* étages associés à *C3*. Cette solution réduit le coût matériel si et seulement si :

$$(L+1)\times SIZE(C3) < L\times(SIZE(C1)+SIZE(C2))$$

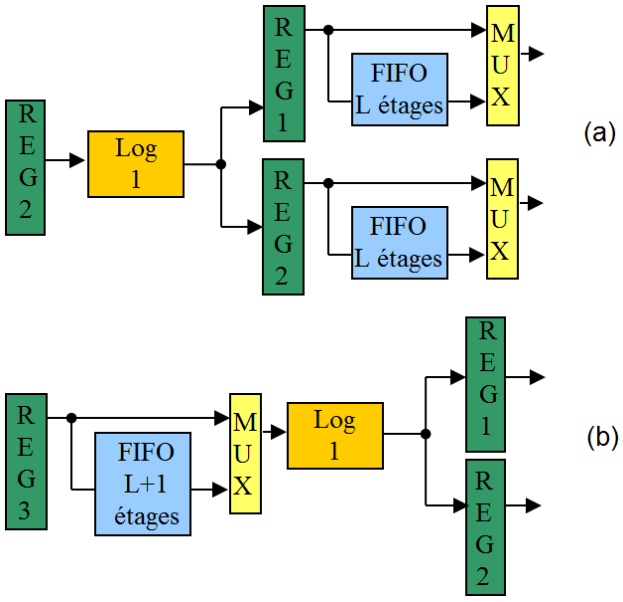

Fig. 3-10 : Optimisation matérielle en jouant sur le nombre de sources

Toutefois cela ne constitue pas le cas le plus général, où les regroupements les plus intéressants ne sont pas situés en entrées des nœuds à remplacer. Mais les expérimentations pratiquées ayant permis de constater qu'entre deux étages successifs d'un circuit, la taille des données stockées est souvent constante, le meilleur moyen de réduire le coût matériel est en général de réduire le nombre de sources. Pour y parvenir, pour chaque nœud de la solution initiale $SEI_k \cup SCI_k$ on établit l'ensemble de leurs prédécesseurs, et on cherche les prédécesseurs communs à plusieurs sources. On peut ainsi regrouper plusieurs FIFO en une seule. Ce principe est d'autant plus efficace lorsque certains de ces prédécesseurs communs appartiennent déjà à l'ensemble $SEI_k \cup SCI_k$.

Dans les circuits à l'architecture irrégulière, le nombre de regroupement possible peut être très important.

Ce principe est illustré par la figure 3-11. Cette figure représente une partie d'un circuit où trois nœuds *s1, s2* et *s3* ont été identifiées comme des sources possibles pour la décontamination de ce circuit. Dans cette figure les sources *s1* et *s2* ont le nœud *e* en commun. Si on souhaite remplacer la FIFO associée au nœud *s1*, il faut faire en sorte que l'information fournie par la FIFO puisse être restaurée. Aussi la FIFO de *s1* doit être remplacée par des FIFO associées aux nœuds *a, d,* et *e,* avec $FIFO(a)=FIFO(s1)+1$, $FIFO(d) = FIFO(s1) + 2$, et $FIFO(e) = FIFO(s1) + 2$. De même la FIFO de *s2* est remplacée par une FIFO associée au nœud *e*, $FIFO(e)=FIFO(s2)+ 2$. Finalement d'une première solution *S*, on obtient une solution $S' = (S \setminus \{s1, s2\}) \cup \{a, d, e\}$, avec $FIFO(e)=Max(FIFO(s1)+2,FIFO(s2)+2),FIFO(a)=FIFO(s1)+1$, $FIFO(d)=FIFO(s1)+2$ et on applique ensuite ce principe à tous les regroupements et à tous les ensembles trouvés possibles. Par exemple ici il y a un autre regroupement : *s1* et *s3* si on considère la solution *S*, ou a et *s3* si on considère la solution *S'*.

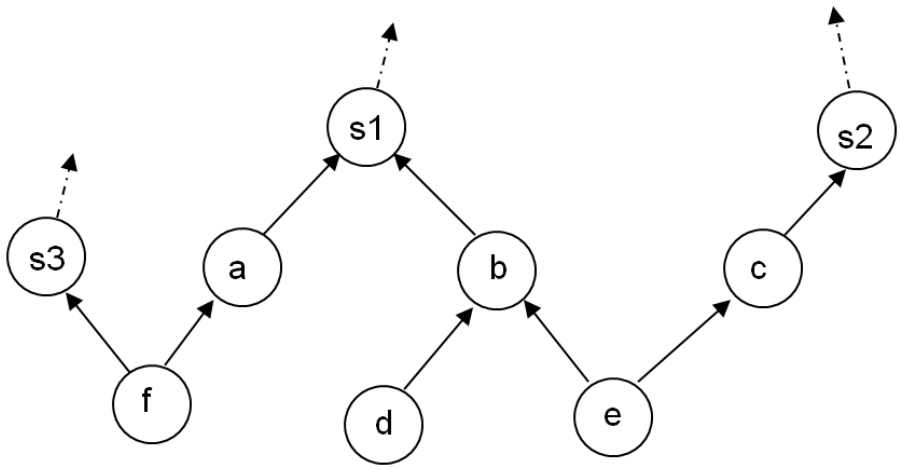

Fig. 3-11 : Regroupement de plusieurs sources

On a donc deux nouvelles solutions :

Solution 1 : $S'' = (S \setminus \{s1, s3\}) \cup \{b, f\}$, $FIFO(f)=Max(FIFO(s3)+1, FIFO(s1)+2)$, $FIFO(b) = FIFO(s1) + 1$.

Solution 2 : $S'''=(S'\setminus\{a,s3\})\cup\{f\}$, avec $FIFO(f)= Max(FIFO(s3)+1, FIFO(s1)+2)$.

La solution conservée sera celle qui minimisera la fonction coût utilisée. L'algorithme basé sur ce principe est détaillé dans l'annexe D. Dans la Figure 3-8, pour $k=3$, nous avons dit que l'ensemble $SEI_k \cup SCI_k$ inclut une FIFO d'un étage pour *N1*, et une FIFO de deux étages pour le nœud *N2.* On suppose que chaque nœud stocke le même nombre de bits. Puisque *N2* est le prédécesseur de *N1*, la FIFO de *N1* peut être retirée, car les données fournies par la FIFO de *N2* sont suffisantes. *N5* n'a pas de prédécesseur, et *N2* a *N3* et lui-même comme prédécesseurs. Donc aucune autre optimisation supplémentaire ne peut être faite.

3.3.5 Cycles de décontamination

La dernière étape de l'algorithme est la description du processus de décontamination. Que la solution choisie soit optimisée du point de vue du coût matériel, ou du temps de décontamination, on considère une solution *S*, et l'ensemble des FIFO associées. Rappelons que dans la section 3.2.3 on a précisé les cycles de décontamination durant lesquels les FIFO sont actives, lesquels sont liés à la taille des FIFO. Les cycles d'horloge sont comptés à partir du moment où les données corrigées sont émises. La décontamination démarre au cycle $k - max(FIFO(s), s \in S) + 1$, et finit au $k^{ème}$ cycle. On rappel également que, lorsqu'un numéro de cycle d'horloge est inférieur à 1, cela signifie qu'il s'agit d'une étape devant précéder l'émission de la donnée corrigée par le code correcteur. Comme précisé dans la section 2.5, tous les composants de stockage sont d'abord bloqués. A *clk=1*, les données corrigées sont fournies par le circuit de correction. A chaque cycle, l'état des bascules débloquées est mis à jour.

Le principe pour déterminer les cycles de décontamination repose sur une idée simple déjà développée dans la section 2.5 : une fonction logique, durant un cycle de décontamination, fournit un résultat valide si toutes ses entrées sont valides. Le résultat valide est alors stocké dans un registre, lequel doit être débloqué durant ce cycle pour enregistrer le résultat. Au cycle suivant, il fournira une donnée valide. Du point de vue de la représentation sous forme de graphe, un nœud *v* est considéré comme valide durant un cycle *clk*, si au cycle *clk–1* tous les nœuds *Ip(v)* sont valides. Une exception est faite pour les nœuds sources. Un nœud source fournit une donnée valide si celui-ci est décontaminé au cycle précédent (pour les sources contaminables), ou si la FIFO associée est active. Dans le cas d'une source contaminable, rappelons que la FIFO associée n'est plus active dès que la source est décontaminée. Les principes sont valables quelque soit la solution considérée. De cette façon on peut déterminer cycle par cycle les FIFO qui sont actives, et les registres qui doivent être débloqués. L'algorithme est donné dans l'annexe E.

Pour la figure 3-8 et *k=3*, *C5* et *C2* sont respectivement associés à des FIFO de 1 et 2 étages. A *clk=1*, *C3* est débloqué. A *clk=2*, *C2, C1* et *C4* sont débloqués. La donnée fournie par la FIFO de *C2* est transmise. A *clk=3, C0* est débloqué. La donnée fournie par la FIFO de *C5* est transmise. La FIFO associée à *C2* ne fournit plus de donnée car *C2* avait été décontaminé durant le cycle précédent. A *clk >3*, le processus est fini.

3.4 Cas d'étude et résultats

L'algorithme décrit dans la section 3.3 a été implémenté à l'aide du logiciel Matlab. L'expérimentation a été faite en premier lieu sur le cas d'étude présenté dans la section 2.8. Lors de cette expérimentation, l'ensemble des mémoires ont été considérées, et pas uniquement celles dédiées à la FFT comme dans le chapitre 2. De plus, le nombre *k* n'a pas été limité au seul cas de figure rencontré lors de ces premiers tests, *k* étant le nombre de cycle d'horloge nécessaire avant de bloquer le circuit. Le but ici est d'évaluer le surcoût de la solution en termes de FIFO et multiplexeur additionnel par rapport à

l'ensemble du circuit, ainsi que le gain apporté par la fonction d'optimisation par rapport à une solution composée uniquement de sources immédiates (solution la plus naïve). Contrairement au chapitre 2 où le surcoût était calculé en fonction de la mémoire protégée, le surcoût de l'implémentation ainsi que les possibilités d'optimisation dépendent de l'architecture du circuit. C'est pourquoi le coût matériel a été évalué ici en fonction du circuit, sans les mémoires RAM. La prise en compte de ces mémoires aurait évidement encore diminué le surcoût relatif de notre solution. Les résultats sont présentés dans le tableau 3-2. La colonne "Surcoût initial" montre le surcoût dû à l'utilisation de la solution en utilisant uniquement les sources immédiates. La colonne "Réduction" montre le gain apporté par la fonction d'optimisation par rapport à la solution composée uniquement de sources immédiates. La colonne "Surcoût" représente l'augmentation matérielle relative de la puce à cause de la solution optimisée.

Pour $k=1$ ou 2, l'ensemble des sources immédiates $SEI_k \cup SCI_k$ et leur FIFO associée constitue une solution optimisée d'un point de vue du matériel et du temps de décontamination. En revanche, dès que $k \geq 3$, la solution constituée de sources immédiates n'est plus la solution optimale, et l'outil réduit le surcoût matériel jusqu'à 15% pour $k=5$. En rapport pour $k=5$, le surcoût des FIFO et multiplexeur de la solution optimisée est de 6,85%, contre 7,90% si on utilise la solution non-optimisée. En revanche la solution n'est plus optimisée d'un point de vue du temps de décontamination, et nécessite jusqu'à 10 cycles d'horloge supplémentaire au total pour faire la décontamination, ce qui n'est pas rédhibitoire, compte tenu de la perte de performance évaluée dans la section 2.7. En effet la perte est à peine accrue : considérons un circuit comportant 60 Mbytes de mémoire embarquée, et un SER de 1000 FIT par Mbit, le nombre total d'erreurs par heure sera de $60 \times 8 \times 1000 \times 10^{-9} = 48 \times 10^{-5}$, soit une erreur présente en mémoire tous les 87 jours (1 FIT = 10^{-9} erreur par heure). Considérons maintenant un nombre de cycles d'horloge relativement grand pour la décontamination d'une erreur (dû à l'utilisation d'une solution optimisée), engendrant une perte de 10 cycles d'horloge tous les 87 jours. Alors pour une fréquence d'horloge de 200 MHz cela provoque une baisse de performance de 10^{-14} %. Ce qui veut dire que même si le temps de décontamination est doublé, il reste insignifiant par rapport à la fréquence d'apparition des erreurs.

Tab. 3-2 : Coût matériel additionnel de l'algorithme appliqué au modulateur OFDM

Circuit	k	Surcoût initial	Réduction	Surcoût final
Modulateur OFDM	1	+ 0,50 %	0 %	+ 0,50 %
	2	+ 2,33 %	0 %	+ 2,33 %
	3	+ 4,16 %	– 7,18 %	+ 3,88 %
	4	+ 6,05 %	– 11,3 %	+ 5,44 %
	5	+ 7,90%	– 15,3 %	+ 6,85 %

Il semble toutefois évident que le surcoût matériel comme la possibilité de le réduire dépend de l'architecture du circuit. En effet, ainsi que nous l'avons dit, l'implémentation est beaucoup plus simple si l'architecture est régulière, alors qu'une architecture irrégulière donnera lieu à des possibilités d'optimisation. Pour vérifier ces

principes nous avons expérimenté l'algorithme sur trois autres circuits provenant de www.opencores.org.

Le premier, noté IP1, est un modulateur OFDM utilisant l'algorithme radix-2, et optimisé pour des FFT sur 64 points. Cette architecture est très régulière, ce qui fait que quelque soit le nombre k considéré elle nécessite peu de sources extérieurs et aucune contaminable. Les résultats sont regroupés dans le tableau 3-3, les colonnes sont exactement les mêmes que pour le tableau 3-2. Les résultats confirment que pour un pipeline parfaitement régulier, il n'est pas possible de trouver une solution moins couteuse que celle constituée par des sources immédiates. D'autre part, le faible nombre de source extérieure conduit à un surcoût même pour *k=5* relativement faible (1,6%).

Tab. 3-3 : Coût matériel additionnel de l'algorithme appliqué au circuit IP1

Circuit	k	Surcoût initial	Réduction	Surcoût final
IP1	1	+ 0,2 %	0 %	+ 0,2 %
	2	+ 0,4 %	0 %	+ 0,4 %
	3	+ 0,8 %	0 %	+ 0,8 %
	4	+ 1,2 %	0 %	+ 1,2 %
	5	+ 1,6 %	0 %	+ 1,6 %

Le deuxième noté IP2, est un encodeur JPEG streamer, comportant un grand nombre de mémoire et de buffer. De plus l'architecture est très irrégulière ce qui conduit à un grand nombre de sources contaminables. L'efficacité de la fonction d'optimisation peut alors se vérifier dans le tableau 3-4. Mis à part pour *k=1*, où aucune optimisation n'est possible, la solution initiale peut être optimisée de manière importante (– 23,1 % pour *k=5*), ce qui résulte en un temps de décontamination de 7 coups d'horloge. De même le surcoût matériel est plus important que dans les exemples précédents, jusqu'à + 11,2% pour *k=5*. L'utilisation d'une solution non-optimisée aurait conduit à un surcoût de +13,8%.

Tab. 3-4 : Coût matériel additionnel de l'algorithme appliqué au circuit IP2

Circuit	k	Surcoût initial	Réduction	Surcoût final
IP2	1	+ 0,50 %	0 %	+ 0,50 %
	2	+ 2,89 %	– 3,22 %	+ 2,80 %
	3	+ 5,65 %	– 6,57 %	+ 5,30 %
	4	+ 9,77 %	– 19,2 %	+ 8,20 %
	5	+ 13,8 %	– 23,1 %	+ 11,2 %

Enfin la troisième est un autre modulateur OFDM qui inclut également un filtre récursif. De fait cette architecture est plus irrégulière que celle de IP1. Elle illustre bien la difficulté de prédire le coût de notre solution, car cette architecture ne permet pas beaucoup d'optimisation matérielle (-3,14% pour *k=5*) ce qui donne un temps de décontamination de 6 coups d'horloge. Pourtant l'utilisation de notre solution implique un surcoût matériel dans les mêmes conditions que les autres architectures le plus important (+ 11,5%).

Tab. 3-5 : Coût matériel additionnel de l'algorithme appliqué au circuit IP3

Circuit	k	Surcoût initial	Réduction	Surcoût final
IP3	1	+ 0,10 %	0 %	+ 0,10 %
	2	+ 2,70 %	0 %	+ 2,70 %
	3	+ 5,60 %	0 %	+ 5,60 %
	4	+ 8,68 %	– 2,10 %	+ 8,50 %
	5	+ 11,9 %	– 3,14 %	+ 11,5 %

Les différences sont assez importantes entre les résultats, en particulier sur la possibilité et la difficulté de trouver une meilleure solution, du point de vue matériel, que celle constituée par les sources immédiates. En fonction du circuit visé par notre solution, et du nombre *k* l'utilisation de la fonction d'optimisation peut être ou ne pas être utile. De manière plus globale, cet outil, même si le temps a manqué pour l'intégrer complètement dans le flot de conception, permet de déterminer automatiquement toutes les ressources nécessaires pour la décontamination d'un circuit aussi complexe soit-il. Les différentes étapes de la machine à états lors de la décontamination sont de même entièrement déterminées, et les composants bloqués ou débloqués sont indiqués avec précision.

3.5 Conclusion

En utilisant un modèle basé sur les graphes orientés, l'implémentation de la solution présentée dans le chapitre 2 au niveau RTL peut être déterminée de manière automatique. Les zones du circuit nécessitant des ressources supplémentaires pour sauvegarder les états précédents des composants de stockage sont identifiées. Ces ressources, sous forme de FIFO et de multiplexeur, sont déterminées avec précision, que ce soit le lieu de leur implémentation, ou la taille de chacune des FIFO. En acceptant de perdre quelques cycles d'horloge supplémentaire lors de la correction d'une donnée erronée, il est également possible de changer l'emplacement de ces ressources afin de réduire le coût matériel supplémentaire. Ce délai supplémentaire est toutefois négligeable si on tient compte de la fréquence d'apparition d'une erreur. En revanche la réduction du coût matériel peut être complexe, et fait appel à une fonction d'optimisation sous-optimale, l'optimisation complète étant trop longue à obtenir. Cette fonction est basée sur le principe de pouvoir réduire le nombre de lieux d'implémentation des ressources en recherchant des prédécesseurs communs aux composants de stockage auxquels une FIFO a été attribuée. Si à l'inverse le temps de décontamination est un critère majeur, il est possible de conserver ces ressources au plus près des zones à décontaminer. Quel que soit la solution retenue, les cycles de décontamination sont déterminés avec précision dans une dernière étape. Chaque algorithme permettant d'effectuer chacune de ces étapes est détaillé en Annexe. Quatre cas d'étude ont été utilisés pour tester notre algorithme, donnant des résultats variables en fonction de leur architecture. Pour les architecture les plus régulière, il n'y a pas de réduction matériel possible de la solution initiale (celle ou les FIFO sont placées au plus près des composants à décontaminer). Pour les

architectures les plus complexes, mis à part pour de faible valeur de *k,* les possibilités d'optimisation peuvent être importante et réduire de 20% les FIFO à ajouter. Au final le coût matériel à payer varie, même pour des valeurs importantes de *k* (5), entre 2% et 11% du circuit. Au final cela permet de restaurer l'état d'une zone contaminée d'une puce électronique lorsqu'une donnée erronée a été lue en mémoire. Le chapitre suivant a pour but d'étendre les propositions précédentes à la restauration d'un état correct de l'ensemble du circuit, quel que soit l'origine de l'erreur, et d'évaluer le coût du système permettant la restauration de cet état.

Chapitre 4. Restauration des états précédents d'un circuit

Dans le deuxième chapitre, nous avons présenté une technique efficace au niveau RTL pour éliminer la pénalité de temps due à l'utilisation d'un code correcteur pour protéger une mémoire embarquée. Cette solution était basée sur la restauration des états précédents de certains composants de stockage du circuit. Dans le troisième chapitre, nous avons présenté l'algorithme permettant d'automatiser l'implémentation de la solution. Cet algorithme permettait d'identifier les zones du circuit nécessitant du matériel supplémentaire. Dans ce chapitre nous allons généralisons ces principes à la correction d'une erreur quelconque dans un circuit. L'idée est de considérer que le circuit est doté d'un système de détection permettant de repérer l'apparition d'une erreur, où qu'elle soit dans le circuit. En restaurant l'état du circuit le cycle précédent l'apparition de l'erreur transitoire, celle-ci devrait être effacée par la réexécution des cycles suivants. Plusieurs solutions sont proposées, et comparées avec les résultats fournis par l'état de l'art.

4.1 Le principe du "retour en arrière"

Contrairement aux chapitres précédents, on considère qu'une erreur peut survenir n'importe où dans le circuit. Le circuit est alors doté d'un système de détection d'erreur en rapport. La détection d'une erreur transitoire peut prendre un ou plusieurs cycles d'horloge. Si l'erreur est issue de la mémoire, le système de correction du code correcteur fournit la donnée corrigée. Si la détection se fait au niveau d'une fonction logique ou d'un registre, on part du principe qu'en ré-exécutant le cycle d'horloge précédent l'apparition de l'erreur (et non pas le cycle précédent l'activation du signal de détection), on restaure l'état correct du circuit au moment où l'erreur est apparue. En effet, on suppose que le système de détection de l'erreur est tel qu'en restaurant l'état du système un nombre x de cycles d'horloge en arrière (x étant un nombre fixe, supérieur au temps de détection d'une erreur $k–1$), le système est dans un état correct. Donc, si l'erreur est bien transitoire, en relançant l'exécution du circuit à partir de ce point, la réexécution doit se faire sans que le signal de détection de l'erreur soit réactivé par la même erreur. Encore faut-il restaurer l'état de l'ensemble du circuit x cycles d'horloge en arrière, les solutions fournies par l'état de l'art étant en générale très coûteuse.

Le principe d'un "retour en arrière" dans un circuit où une erreur transitoire a été détectée est donc de revenir dans un état correct, précédent l'apparition d'une erreur (figure 4-1). Si l'erreur était bien transitoire, celle-ci devrait ne pas réapparaître lors de la réexécution des cycles suivants. On peut également parler de "masquage" de l'erreur grâce à une technique de "redondance temporelle partielle". En effet, une erreur a bien été détectée, même celle-ci va être éliminée au prix de quelques cycles d'horloges supplémentaires pour ré-exécuter les cycles d'horloge concernés par l'erreur. D'où également le terme de "partielle", par opposition à une "redondance temporelle complète" qui consisterait à ré-exécuter à chaque fois une ou plusieurs fois les même opérations. L'équivalent matériel de cette technique, est la redondance matérielle déjà présentée (DMR, TMR, code…), ou la réexécution se fait en parallèle sur plusieurs circuits différents.

Dans le premier chapitre nous avons déjà dressé un état de l'art des systèmes de détection d'erreurs dans les circuits. Dans la majorité des cas, ces systèmes de détection doivent être combinés avec un autre système destiné à restaurer un état correct du circuit à partir duquel le fonctionnement normal peut reprendre. Dans ce travail nous avons donc opté pour une réexécution du cycle d'horloge durant lequel une erreur est apparue. Cela peut se faire de deux manières différentes. La première, a priori la plus simple, est de considérer que le signal de détection d'erreur est actif $k–1$ cycles d'horloge après l'apparition d'une erreur. Alors il suffit de restaurer l'état précédent du circuit k cycles d'horloge avant la détection de l'erreur pour réparer l'erreur. Une autre possibilité consiste à utiliser des points de sauvegarde, c'est-à-dire des cycles d'horloge particuliers, présents à intervalle régulier durant l'exécution du circuit. A chacun de ces cycles, l'état complet du circuit est sauvegardé. Lorsqu'une erreur est détectée, le système redémarre depuis le

dernier état correct sauvegardé [SS92][Pra92]. Une implémentation logicielle pour les architectures à base de processeurs de cette technique ne nécessite pas de modification matérielle importante. Toutefois le coût final est important puisque selon [RRV+02], cela nécessite beaucoup d'espace mémoire (5× pour la détection d'erreur, 6× pour la détection et la correction) et une pénalité de temps importante (3× pour la détection, et 4× pour la détection et la correction). Une autre solution valable dans les processeurs, est de ré-exécuter l'instruction durant laquelle l'erreur est apparue, ainsi que toutes les instructions suivantes.

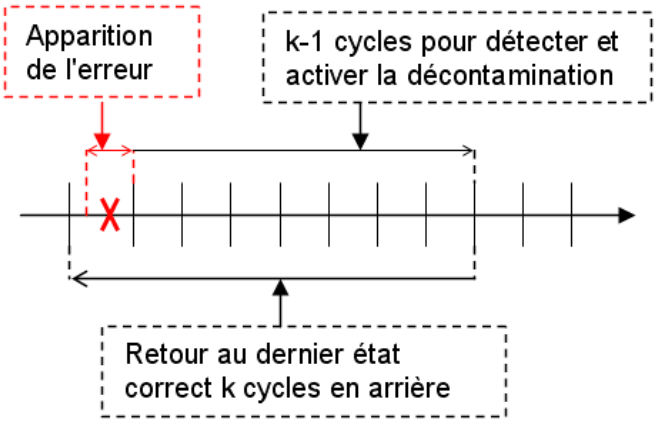

Fig. 4-1 : Principe du "retour en arrière"

Autant que nous le sachions, la seule approche au niveau matériel pour effectuer un "retour en arrière", qui puisse être utilisée dans tous types d'architecture matérielle synchrone est détaillée dans [TTR88][TT89]. Leur méthode consiste à traiter individuellement chaque composant de stockage du circuit. Pour effectuer un "retour en arrière" de k cycles d'horloge (à supposer que la détection de l'erreur prenne $k–1$ cycles d'horloge), tous les composants de stockage (latch, flip-flop registre ou RAM) doivent revenir à leur $k^{ème}$ état précédent. Deux techniques différentes sont utilisées en fonction du composant de stockage considéré. Pour le stockage de masse, RAM ou banc de registre (dans les processeurs notamment), la solution consiste à décaler de k cycles d'horloge l'écriture des données en mémoire. Ces données, et les adresses auxquelles elles seront stockées k cycles d'horloge plus tard sont, respectivement, stockées temporairement dans une FIFO et une CAM. Comme une donnée n'est écrite en mémoire que k cycles d'horloge plus tard, l'état de la mémoire dans les k cycles précédant l'apparition de l'erreur est préservé. En effet durant les k derniers cycles seront écrits en mémoire des données qui ont été introduites dans le système FIFO/CAM plus de k cycles d'horloge auparavant. Lors de la lecture d'une donnée en mémoire, l'adresse est comparée à celles stockées dans la CAM. S'il y a une ou plusieurs correspondances, la dernière donnée écrite dans la FIFO/CAM correspondant à cette adresse est utilisée lors de la lecture, sinon la donnée est lue à l'adresse correspondante de manière classique. Ce système présente l'avantage, pour les systèmes de stockage très volumineux, d'avoir proportionnellement un système de sauvegarde peu coûteux. En effet au lieu de stocker

une copie de tous les mots enregistrés en mémoire, il est moins coûteux de sauvegarder les *k* dernières opérations.

En revanche, toujours dans [TTR88][TT89] chacun des composants de stockage isolé est associé à un banc de registre de taille *k* (et non pas à une FIFO comme dans les chapitres précédents) qui sauvegarde les *k* derniers états du composant. Le principal avantage est que la restauration du $k^{ème}$ état précédent prend un seul cycle d'horloge, puisque tous les composants de stockage isolés ont leur propre banc de registre associé (figure 4-2). De plus leur système autorise un "retour en arrière" de *k* cycles au plus, c'est à dire qu'il permet, en un seul cycle, un "retour en arrière" de *k* cycles, *k–1* cycles, *k–2* cycles… C'est-à-dire que le banc de registre est doté d'un système d'adressage et de lecture pour que la donnée souhaitée soit disponible en un cycle. De plus les auteurs ont supposé qu'une erreur dans le circuit pouvait ne pas être isolée, et ont doté chacun de ces bancs de registres d'un système de bit de parité. La conséquence principale est que le coût matériel d'une telle solution est forcément important. Les auteurs évaluent eux-mêmes le surcoût en surface d'une telle solution pour *k=4* à environ six fois la surface de chaque composant de stockage isolé.

Nous allons proposer ici une solution qui privilégiera à l'inverse la réduction du coût matériel au détriment du nombre de cycles d'horloge nécessaire au rétablissement de l'état du circuit *k* cycles en arrière. Ce principe existe déjà dans les processeurs sous la forme de réexécutions d'instructions. Cette solution, comme dans les chapitres précédents, est donc surtout dédiée aux architectures qui ne sont pas orientées processeurs. Deux types de techniques seront proposés : un retour arrière de *k* cycles d'horloge, *k* étant un nombre fixé, et une solution basée sur un système avec des points de sauvegarde.

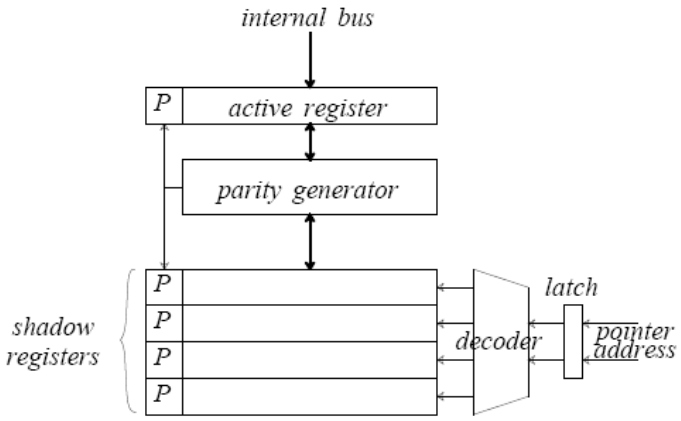

Fig. 4-2 : Protection des composants de stockage isolés [TT89]

4.2 Sauvegarde d'états pour les registres durant k cycles

Cette partie de la présentation est dédiée à l'exposé de notre solution pour la préservation, durant un nombre *k* de cycles d'horloge, des états des composants de stockage isolés. Comme nous l'avons déjà expliqué dans les chapitres précédents les *k* états précédents d'un composant de stockage isolé peuvent être sauvegardés par une FIFO de taille *k*. Ainsi alors que l'état d'un registre est mis à jour à chaque cycle, l'état précédent est conservé pendant *k* cycles. Donc à chaque cycle, le premier étage (entrée de la FIFO) contient l'état précédent, et le dernier étage (sortie de la FIFO) contient l'état du registre *k* cycles auparavant (voir figure 4-3 (a)).

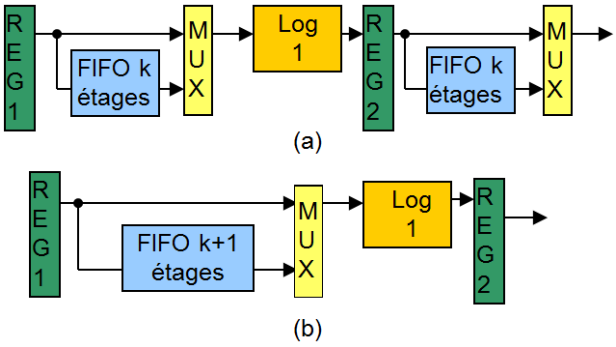

Fig. 4-3 : Protection des composants de stockage isolés ; ressources mise en en commun

Lorsque cet état est requis, il peut être fourni immédiatement par un MUX, le cycle suivant le signal de détection de l'erreur. Le $k^{ème}$ état précédent est donc restauré et utilisé dès le cycle suivant. On considère maintenant deux registres séparés par une fonction logique. Au lieu d'utiliser une autre FIFO de *k* étages pour protéger les *k* derniers étages du registre *REG2*, les *k* derniers états de *REG2* peuvent être sauvegardés grâce à une FIFO de *k+1* étages associée à *REG1* (figure 4-3 (b)). Le dernier étage de la FIFO contient alors le $k+1^{ème}$ dernier état de *REG1*. En ré-exécutant la *k+1* dernière opération de *Log1*, on restaure ainsi le $k^{ème}$ dernier état de *REG2*. Donc au lieu d'utiliser deux FIFO de *k* étages, on utilise une seule FIFO de *k+1* étages. La contrepartie est que l'on utilise un cycle d'horloge supplémentaire avant de pouvoir fournir le $k^{ème}$ état précédent. On peut facilement étendre ce principe à un pipeline de *P* étages. La FIFO associée au premier étage peut sauvegarder les *k* derniers états de l'ensemble des *P* étages du pipeline, en utilisant une FIFO de taille *k+P–1*. La contrepartie est que le $k^{ème}$ état précédent du pipeline est restauré en *P* cycles d'horloge.

De la même manière que dans le chapitre précédent, où le lieu d'implémentation des FIFO était fonction des irrégularités du circuit, le discours précédent fonctionne uniquement tant que le pipeline reste régulier. Dans ce cas de figure, une FIFO de k+*P*–1 étages associée aux premiers étages des pipelines de taille *P* permet d'effectuer le "retour en arrière". Toutefois si le nombre d'étages *P* est très grand, la restauration de l'état précédent peut être très long. Il peut être éventuellement utile, de limiter le nombre d'étages de pipeline auxquels une FIFO est liée. Par conséquent, une FIFO de *k+p–1*

étages peut être utilisée pour préserver le $k^{ème}$ état précédent d'un pipeline de p étages $(p<P)$. Une autre FIFO sera ensuite associée à l'étage $p+1$, laquelle sera chargée de mémoriser le $k^{ème}$ état précédent de cet étage, et des étages suivants (Figure 4-4). Par analogie au chapitre précédent, on appellera un registre associé à une FIFO, une source.

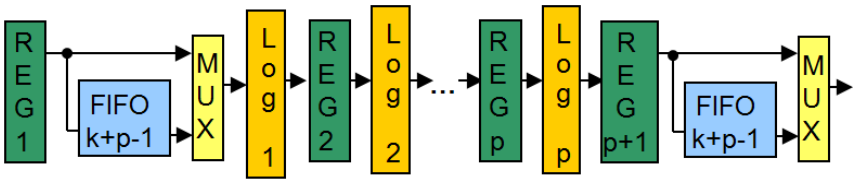

Fig. 4-4 : Limitation du temps de décontamination, par limitation du nombre *p*

L'avantage de ce principe, est que cela permet d'adapter facilement cette méthode à des architectures irrégulières. Le principe de base est de diviser une architecture irrégulière en plusieurs pipelines réguliers. Chacun des premiers registres de ces pipelines réguliers est alors utilisé comme source pour le reste du pipeline. La figure 4-5 illustre un segment de circuit qui contient tous les types d'irrégularités que l'on peut rencontrer dans un circuit. On peut noter que les limites des parties régulières (délimitées par les contours rouges), correspondent à des circuits logiques qui possèdent plusieurs entrées. Pour rappel, dans les chapitres précédents, nous avions signalé qu'une source ne pouvait exister que si une fonction logique d'un circuit (ou un nœud si on considère la représentation sous forme de graphe orienté) possédait plusieurs entrées.

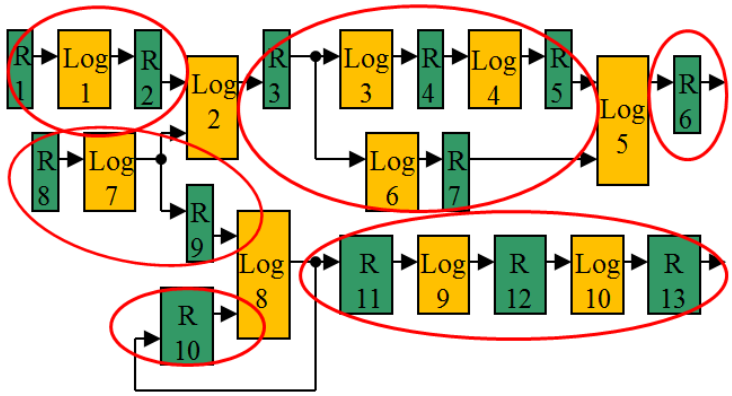

Fig. 4-5 : décomposition du circuit en pipeline régulier

Le bloc *Log2* délimite deux parties régulières, car les entrées proviennent de deux sources différentes, *R1* et *R8*, dont les distances qui les séparent de *R3* sont différentes (distance au sens défini dans le chapitre précédent). Le circuit entre *Log 2* et *Log 5* est régulier car le chemin *R3, R4, R5* est indépendant du chemin *R3, R7*. Toutefois cette partie régulière du circuit ne va pas au-delà, jusqu'à *R6*, car le chemin *R3, R7* et *R6* est plus court que le chemin *R3, R4, R5* et *R6*. A cause de la boucle, *R10-log8-R10, R10* est une partie régulière qui ne comporte qu'un seul composant. Chaque premier étage de chaque partie régulière est associé à une FIFO pour pouvoir effectuer le "retour en arrière" de cette partie régulière. Dans la figure 4-5 les composants qui ont une FIFO

associée sont *R1, R3, R6, R8, R10* et *R11*. La taille de chaque FIFO est calculée en fonction de *k* et du nombre d'étages qui compose la partie régulière (par exemple *p=3* pour *Reg3*). En ajoutant une FIFO à chaque nœud source identifié, le "retour en arrière" de chaque partie régulière peut être effectué. Cela permet de s'affranchir des problèmes de dépendance de données, car chaque source suffit à restaurer le contexte du pipeline auquel elle est associée. La restauration est effectuée en un nombre de cycle $p_M=Max(p, p$ taille des pipelines réguliers).

Pour qu'il n'y ait pas de confusion, on différenciera :

- **La phase normale** : phase d'exécution du circuit classique en l'absence d'erreur. Lorsqu'une erreur est détectée, ce sont les *k* derniers cycles de cette phase, précédant l'activation du signal de détection d'une erreur, qui devront être ré- exécutés.
- **La phase de réexécution** : phase pendant laquelle les *k* derniers cycles de la phase normale doivent être effectués.
- **La phase de restauration d'état** : cette phase est destinée à restaurer le contexte du $k^{ème}$ cycle précédant la phase d'activation de l'erreur. Elle se situe donc entre la fin (pour cause d'activation du signal d'erreur) de la phase normale et le début de la phase de réexécution (figure 4-6).

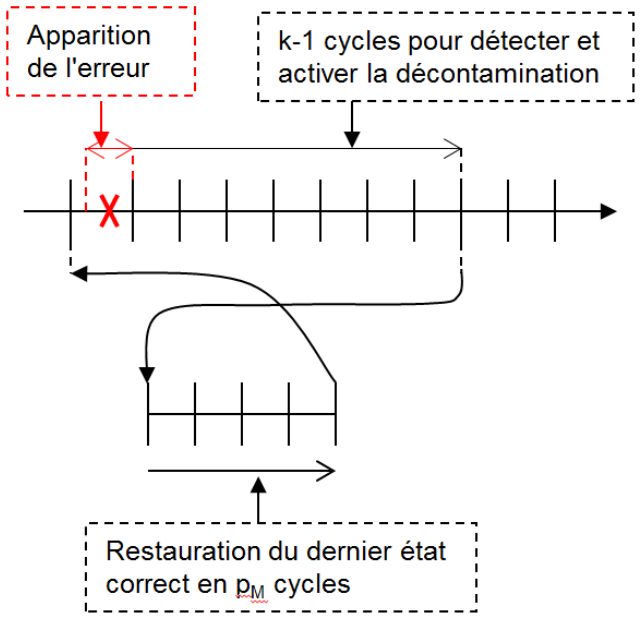

Fig. 4-6 : Restauration en p_M cycles du $k^{ème}$ état précédent

4.3 Sauvegarde d'états pour les mémoires

Cette partie est destinée à présenter le mécanisme de préservation des états des composants de stockage de masse. Pendant les cycles de la phase de réexécution, les fonctions logiques fourniront les mêmes adresses de lecture et d'écriture que durant la phase régulière. De plus, à chaque opération d'écriture de la phase de réexécution, les

fonctions logiques fourniront les données correctes qui auraient dues être fournies à la mémoire durant la phase normale (l'une de ces données pouvait être corrompue durant la phase normale). En même temps, lors des opérations de lecture de la phase de réexécution, les mémoires doivent fournir les données qu'elles auraient dû fournir durant la phase normale. Cependant, durant les *k* derniers cycles de la phase normale, les données de certaines adresses mémoires ont pu être modifiées, et donc les données qui auraient dû être fournies lors de la réexécution peuvent être effacées. Par conséquent, les mémoires doivent disposer d'un système particulier pour préserver les dites données. Ce système sera détaillé dans son fonctionnement pour les phases normales et de réexécution. Puis seront exposés les ajustements à fournir pour la phase de restauration du contexte.

En plus du système chargé de la détection d'erreurs durant le fonctionnement du circuit, les données peuvent être stockées dans une mémoire pendant un long moment. Par conséquent, comme dans les chapitres précédents, une erreur peut survenir dans la mémoire. La simple réexécution des *k* derniers cycles ne va alors pas suffire à corriger l'erreur, et le système de stockage doit disposer d'un système de détection/correction qui lui est propre, tel qu'un code correcteur. Supposons dans un premier temps qu'une erreur n'est pas intervenue dans la valeur des adresses de lecture ou d'écriture lors des *k* derniers cycles d'exécution du circuit. La solution que nous proposons permet de répondre aux cas suivants.

Cas a : Durant les *k* cycles précédant la détection de l'erreur, et qui devront être répétés, une lecture peut être faite à une adresse en étant suivie (toujours durant ces *k* cycles) par une opération d'écriture à la même adresse. Par conséquent, durant la phase de réexécution, la donnée qui doit être fournie au circuit ne peut pas l'être simplement en relisant la même adresse mémoire, car celle-ci a été modifiée par l'opération d'écriture. La donnée lue doit donc être sauvegardée dans un mécanisme particulier, et doit être ensuite fournie lors de la réexécution lorsque l'adresse de lecture correspondante est fournie à la mémoire.

Cas b : Supposons qu'une donnée erronée soit écrite en mémoire, et qu'une lecture à la même adresse se fasse avant la détection de l'erreur. La donnée fournie lors de la lecture (qui ne peut pas être corrigée par un système ECC associé à la mémoire) est erronée. De plus on ne peut pas réutiliser la donnée sauvegardée dans le mécanisme de sauvegarde car celle-ci est erronée. D'un autre côté, la donnée fournie à la mémoire pendant la réexécution, elle, sera correcte. C'est donc la donnée sauvegardée en mémoire pendant la phase de réexécution qui doit être utilisée. Pour simplifier l'implémentation on utilise les options suivantes.

Pour les opérations d'écriture :

- pendant la phase normale les données écrites en mémoire ne sont pas sauvegardées dans le mécanisme de sauvegarde ;

- pendant les phases de réexécution on effectue uniquement des écritures dans la mémoire, et aucune dans le mécanisme de sauvegarde.

Pour les opérations de lecture :

- chaque donnée lue durant la phase normale est sauvegardée dans le mécanisme de sauvegarde ;
- durant la phase de réexécution, si une opération de lecture est faite à une adresse, précédée d'une opération d'écriture à la même adresse mémoire, la donnée fournie est celle de la mémoire. Si aucune écriture de ce genre a été faite (ou alors après la lecture à cette même adresse et avant le début de la phase de réexécution), on lit la donnée stockée dans le mécanisme de sauvegarde.

Pour cela on a besoin d'un indicateur (drapeau) qui indique si la donnée sauvegardée dans le mécanisme de sauvegarde est valide, pour savoir quelle donnée doit être utilisée. Toutefois il est difficile de prédire quand une détection d'erreur va survenir, et donc de mettre à jour le drapeau pendant la phase normale. Par conséquent on préférera mettre à jour le drapeau pendant la réexécution des cycles. Pour mettre à jour le drapeau, il faut déterminer si l'opération effectuée lors d'un cycle correspond à une opération effectuée précédemment. On peut, pour cela, utiliser une CAM ; pour notre solution, chaque zone de la CAM est composée de trois champs correspondants aux données suivantes.

- **Le champ adresse** stocke l'adresse d'une opération effectuée en mémoire durant un cycle (lecture ou écriture). Ces adresses, suivant l'implémentation choisie, doivent pouvoir être comparées aux adresses de lecture ou d'écriture à chaque cycle. Ces comparaisons sont effectuées pendant la phase de réexécution.
- **Le champ de données** sauvegarde les données d'une opération faite en mémoire. Ces données sont utilisées seulement pendant la phase de réexécutions.
- **Le champ drapeau** ne possède qu'un seul bit qui indique si une donnée stockée dans la CAM est utilisable durant la phase de réexécution.

La CAM utilisée ici possède deux mécanismes d'adressage. Le premier, compare une adresse avec toutes celles stockées dans le champ adresse de la CAM. Lorsqu'une correspondance existe, la donnée stockée dans le champ de données correspondant au champ adresse peut être utilisée pour l'opération courante. Une autre possibilité est d'utiliser l'emplacement utilisé le moins récemment. Pour cela on utilise un mécanisme d'adressage séquentiel qui, de manière cyclique pointe vers chacun des emplacements. A chaque cycle d'horloge un nouvel emplacement est choisi. Par conséquent, dans une CAM contenant k mots, chaque mot est renouvelé tous les k cycles. Le point intéressant dans cette implémentation est que, lorsque plusieurs opérations de lecture ou d'écriture

sont faites à la même adresse pendant les *k* derniers cycles, la même adresse est stockée à plusieurs reprises dans la CAM. Cela évite de détruire la donnée précédente en réécrivant dans le même emplacement. A ce moment-là, lorsque le premier mécanisme d'adressage indique plusieurs correspondances avec la même adresse, on utilisera en priorité la moins récente. Plusieurs implémentations/algorithmes de cette CAM ainsi décrites sont possibles.

4.3.1 Implémentation/algorithme 1

Dans cette implémentation (figure 4-7), le champ adresse stocke seulement les adresses des opérations de lecture. Le champ des données sauvegarde seulement les données lues en mémoire.

Pour chaque cycle de la phase normale :

- Si aucune lecture n'est effectuée durant ce cycle, le drapeau de l'emplacement actuel de la CAM est mis à 0.
- Si une lecture est faite durant ce cycle, une opération d'écriture est effectuée à l'emplacement actuel de la CAM. L'écriture consiste à stocker l'adresse de lecture, la donnée lue, et mettre le drapeau correspondant à 1.

Pour chaque cycle de la phase de réexécution :

- Pour une opération de lecture, le drapeau de l'emplacement actuel est examiné. S'il est égal à 1, la donnée est lue, si le drapeau est à zéro on lit la donnée en mémoire.
- Comme précisé précédemment, chaque opération d'écriture est effectuée dans la mémoire. Mais en même temps, l'adresse d'écriture sera comparée en parallèle avec chaque champ adresse de la CAM. S'il y a une ou plusieurs correspondances, les drapeaux correspondants seront mis à 0. De cette manière, on sait que la donnée correcte figure en mémoire et non dans la CAM.

4.3.2 Implémentation/algorithme 2

Dans cette implémentation (figure 4-7), les champs de données sont séparés des champs adresse et drapeau. Les données peuvent par conséquent être stockées dans une simple FIFO de taille *k*, tandis que les adresses et les drapeaux seront stockés dans une CAM.

Pour chaque cycle de la phase normale :

- les données lues sont stockées dans la FIFO.

Au début de la phase de réexécution, tous les drapeaux de la CAM sont mis à 0.

Pour chaque cycle de la phase de réexécution :

- lorsqu'une écriture est faite en mémoire, l'adresse de l'opération est écrite dans la CAM, et le drapeau correspondant est mis à 1.
- lorsqu'une lecture en mémoire doit être effectuée, l'adresse de lecture est comparée en parallèle avec tous les champs adresse de la CAM. S'il y a une correspondance et que le drapeau associé est à 1, la lecture est faite en mémoire, sinon la donnée est lue à partir de la FIFO. Durant cette phase, la FIFO est dépilée cycle après cycle.

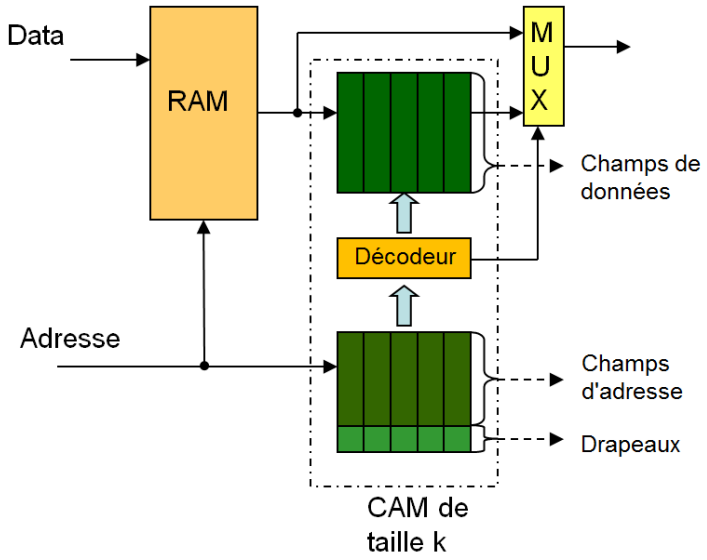

Fig. 4-7 : Schéma général des implémentations 1 et 2

4.3.3 Implémentation/algorithme 3 provenant de l'état de l'art

La troisième solution est celle proposée dans [TTR88][TT89] que nous avons déjà évoquée (Fig 4-8). Durant la phase normale, lors d'une opération d'écriture, l'adresse d'écriture et la donnée correspondante sont écrites dans la CAM, et les données ne sont pas écrites en mémoire. Les données ne sont écrites en mémoire que k cycles d'horloge plus tard. Par conséquent l'état de la mémoire est préservé durant k cycles. La contrepartie, contrairement aux deux implémentations précédentes, est que la CAM doit être utilisée pendant la phase normale et pendant la phase de réexécution. En effet, dans le cas général, on ne peut pas prédire quand les données écrites en mémoire devront être lues. Par conséquent à chaque cycle, l'adresse de lecture est comparée à toutes les adresses stockées dans la CAM durant la phase normale. S'il y a une correspondance, le champ de données le plus récent et qui correspond à l'adresse de lecture est utilisé. Durant la phase de réexécution, si la réexécution se fait sur les $k'<k+1$ derniers cycles d'horloge, alors les k' dernières opérations effectuées sur la CAM sont marquées comme invalides, et leur drapeau est donc mis à 0. Lorsqu'une opération d'écriture se produit pendant la phase de réexécution, l'adresse et la donnée correspondante sont écrite dans la CAM, et le drapeau correspondant est mis à 1. Lors d'une opération de lecture pendant la

phase de réexécution, l'adresse de lecture est comparée uniquement aux emplacements dont le drapeau est marqué comme étant valide.

On peut remarquer qu'entre les trois implémentations proposées, seul l'algorithme d'utilisation change. La taille des données stockées en revanche reste la même : *k* adresses (de lecture ou d'écriture), *k* données (lues ou écrites en mémoire) et *k* drapeaux. Ce surcoût dépend de la taille du composant de stockage de masse à protéger et du nombre *k*. Les schémas ont été fait ici avec des mémoires RAM, mais ils seraient les mêmes avec un banc de registre. Toutefois le nombre de mots stockés dans un banc de registre étant en général plus faible que dans une mémoire RAM, le surcoût est en proportion plus important pour la protection d'un banc de registre. La figure 4-9 est tirée de [TTR88], et donne en fonction du nombre de mots stockés dans le banc de registre (File), et de la taille de la sauvegarde en abscisse (FIFO), le surcoût de cette technique.

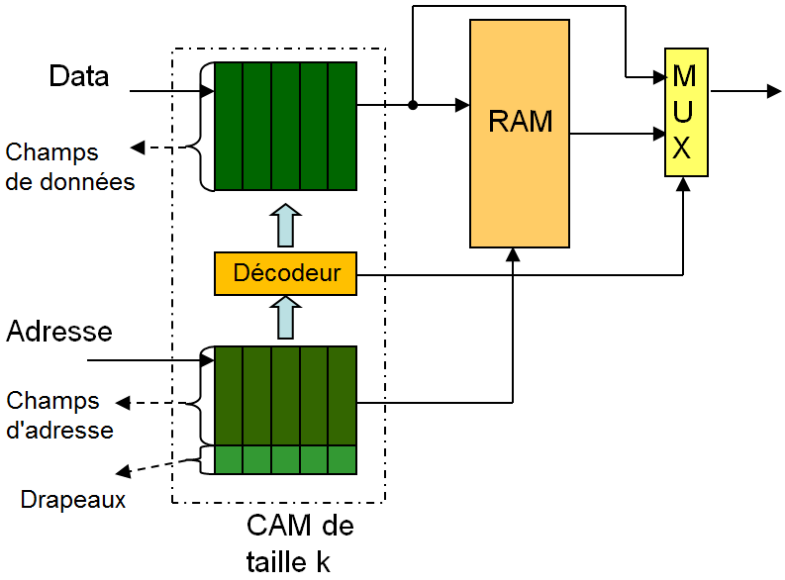

Fig. 4-8 : Schéma général de l'implémentation 3

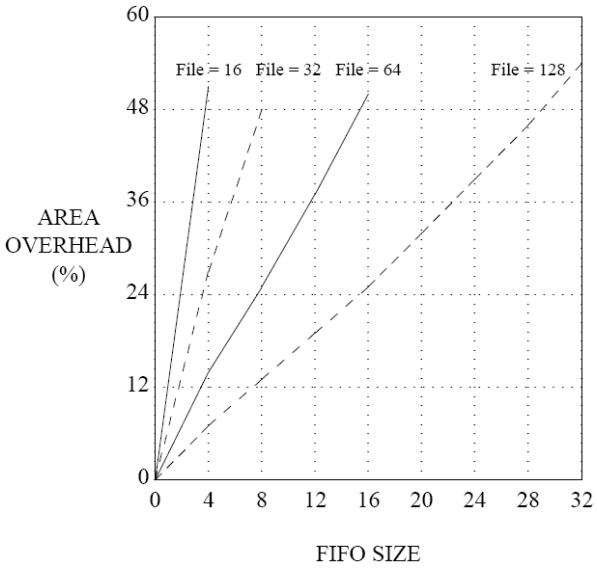

Fig. 4-9 : Surcoût dû au système FIFO/CAM pour différente taille de banc de registre [TTR88]

4.3.4 Gestion des erreurs d'adressage

Jusqu'à présent nous avons supposé qu'aucune erreur d'adresse de lecture ou d'écriture ne se produisait. Nous allons aborder le comportement des trois implémentations proposées lorsqu'une erreur dans l'adresse de lecture se produit, et lorsqu'une erreur d'écriture se produit.

Erreur de lecture : une erreur dans l'adresse de lecture signifie que la donnée a été lue à la mauvaise adresse, et donc que la mauvaise donnée a été fournie au circuit. Dans l'implémentation 1 et 2, cela implique qu'une donnée erronée a été enregistrée dans la CAM. Pour pouvoir corriger cette erreur, il faut pouvoir lire la donnée correcte en mémoire après détection de l'erreur. Ce n'est toutefois pas possible si après l'opération de lecture, une opération d'écriture est effectuée à l'adresse qui aurait dû être lue. A ce moment-là la donnée qui aurait dû être lue n'est plus disponible ni en mémoire ni dans la CAM/FIFO. On peut donc imaginer un système de détection d'erreurs associé à l'adressage qui soit plus rapide que le ou les systèmes contrôlant le reste du circuit pour éviter ce cas. Dans ce cas de figure, et comme on connaît le cycle durant lequel l'erreur s'est produite, il suffit, que ce soit pour l'implémentation 1 ou 2, de prendre la donnée provenant de la mémoire et non celle stockée dans la CAM, pour corriger l'erreur. Dans l'implémentation 3 en revanche, la simple réexécution des k derniers cycles suffit à restaurer le bon contexte, la bonne donnée étant présente en mémoire (on rappelle que toutes les opérations d'écriture sont décalées de k cycles).

Erreur d'écriture : Si une écriture est effectuée à la mauvaise adresse, il n'est pas possible dans les implémentations 1 et 2 de restaurer la bonne donnée. Celle-ci a été effacée, et ne figure pas a priori dans la CAM/FIFO. En revanche, dans l'implémentation 3, les opérations d'écriture ayant été retardées, et celles-ci étant annulées en cas de détection d'une erreur, on évite de faire une mauvaise opération en mémoire.

En conclusion l'implémentation 3 semble être la seule à pouvoir prendre en compte les erreurs d'adressage. En revanche nous n'avons parlé ici que du mécanisme fonctionnant lors de la phase normale et la phase de réexécution, destiné à refaire les k dernières opérations du circuit. Or dans la solution que nous proposons, il faut auparavant faire une opération de restauration de l'état du circuit k cycles avant que le signal de détection ne devienne actif. Nous allons finalement voir qu'à cause de cette phase de restauration de contexte, l'implémentation 3 conduit à un surcoût matériel plus important.

4.3.5 Une mémoire comme premier composant d'un pipeline

Considérons une mémoire connectée en sortie à un pipeline de p étages. Tout comme pour les registres sources dont nous avons parlé dans la section 4.3, il faut d'abord restaurer le contexte correct du pipeline auquel la sortie de la mémoire est connectée avant de refaire les k dernières opérations. Durant cette phase, la mémoire n'est pas active, d'abord parce que les adresses des données lues $k+1,\ k+2,\ldots\ k+p-1$ cycles auparavant ne sont pas forcément disponible, ensuite parce que les données lues en

mémoire lors de ces cycles sont correctes. En effet ce n'est que pendant le $k^{ème}$ cycle précédent l'activation du signal de détection que l'erreur est apparue. On peut donc sauvegarder, comme pour les registres sources, ces données dans une FIFO de $k+p-1$ étages.

Cependant dans les implémentations 1 et 2 les k dernières données lues sont préservées dans la CAM. Il suffit donc dans ces deux implémentations de stocker les données lues dans la CAM, puis après k cycles, de les stocker dans une FIFO de $p-1$ étages qui les conservera encore pendant $p-1$ cycles. Au final elles seront donc bien conservées durant $k+p-1$ cycles (figure 4-10). On insistera sur le fait que contrairement au système de FIFO/CAM destiné à ré-exécuter les k derniers cycles, la FIFO de $p-1$ étages ne conserve que des données, mais ne conserve ni adresse, ni drapeau. Cela n'est toutefois pas possible dans l'implémentation 3, car le système de FIFO/CAM ne conserve que les opérations d'écriture en mémoire retardées de k cycles, et ne conserve pas les données lues. Il faut donc pour l'implémentation 3 une FIFO de $k+p-1$ étages qui stockera les données lues, en plus du système de FIFO/CAM (figure 4-11).

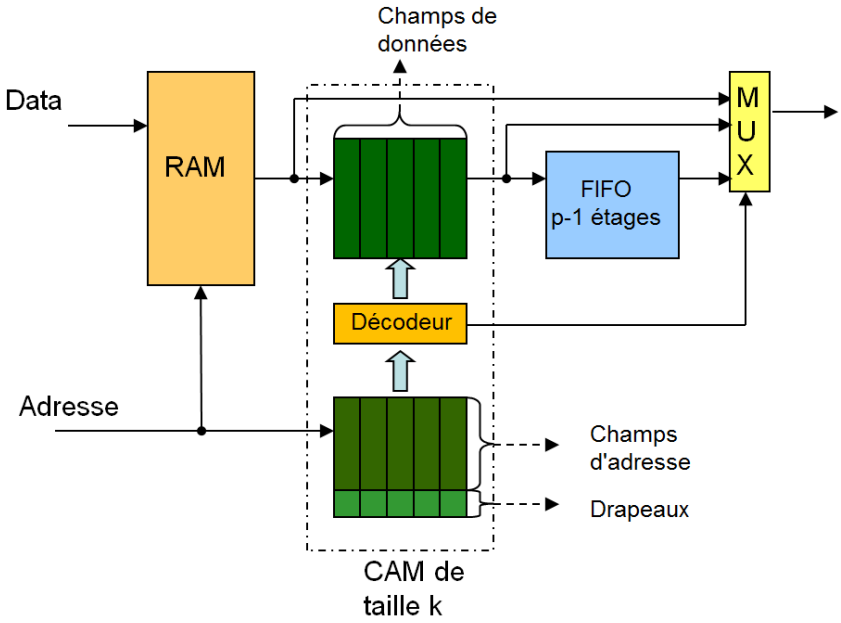

Fig. 4-10 : Implémentation 1 et 2 du système FIFO/CAM avec restauration de contexte

4.4 Implémentation avec check-point

Dans l'approche précédente, la phase de réexécution pouvait démarrer à n'importe quel cycle. Une autre possibilité est d'avoir des points de sauvegarde régulièrement répartis pendant le fonctionnement normal. On considère que la phase normale est divisée en plusieurs *ensembles de cycles (EC)*. Le premier cycle d'un *ensemble de cycles EC* est utilisé comme point de départ pour une éventuelle réexécution si jamais une erreur se produit pendant *l'ensemble de cycles EC* considéré. Le premier cycle de réexécution étant déterminé, il n'est pas nécessaire de faire une sauvegarde en continu de l'ensemble des états des composants de stockage, comme c'était le cas dans l'implémentation précédente.

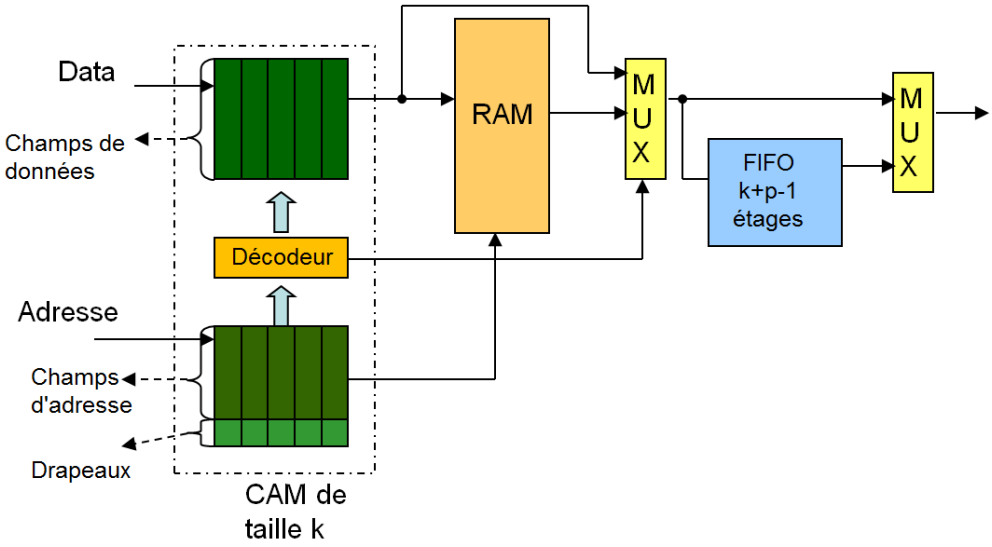

Fig. 4-11 : Implémentation 3 du système FIFO/CAM avec restauration de contexte

4.4.1 Mécanisme de sauvegarde pour les registres

Pour les registres, pendant la phase normale (en dehors des phases de réexécution), on peut sauvegarder seulement la donnée correspondant à ce premier cycle d'un *ensemble de cycles* EC pour chacun des registres. Le principal avantage, c'est qu'en utilisant le principe décrit précédemment, on peut utiliser une FIFO de p étages au lieu d'une FIFO de $k+p-1$ étages pour protéger un pipeline de p étages.

Un inconvénient de cette approche, est que si une erreur se produit dans le dernier cycle d'un *ensemble de cycles EC1*, la détection sera effective k cycles d'horloge plus tard, alors que le nouvel *ensemble de cycles EC2* aura débuté. Cela signifie que la FIFO sauvegardant les états pour la réexécution de ce nouvel *ensemble de cycles EC2* est corrompue. D'autre part c'est le contexte du premier cycle de *l'ensemble de cycles* précédent (*EC1*, celui pendant lequel l'erreur s'est produite) qu'il faudrait restaurer pour que la réexécution soit correcte. Par conséquent, il faut conserver la sauvegarde précédente, jusqu'à ce que l'on soit sûr que la sauvegarde de *l'ensemble de cycles* courant soit correcte. Cela implique qu'il faut que chaque registre source lié à un pipeline de p étages soit associé à deux FIFO de p étages (figure 4-12). Durant une phase normale, l'une des deux FIFO est active durant les $p-1$ derniers cycles d'un *ensemble de cycles*, et le premier cycle de *l'ensemble de cycles* suivant (sauvegarde du premier état du registre source). Cela constitue les données pour la réexécution de *l'ensemble de cycles EC1* enregistré dans la *FIFO1*. Les données associées à *l'ensemble de cycles EC2* seront enregistrées de la même façon, mais dans la *FIFO 2*. Lorsque les données associées à *l'ensemble de cycles EC3* seront enregistrées, alors les données associées à *l'ensemble de cycles EC1* dans la *FIFO 1* seront effacées et remplacées par les données de *l'ensemble de cycles EC3*, et ainsi de suite (figure 4-13). Comme le nombre k de cycle d'horloge nécessaire avant que la détection ne se fasse est connu, on sait, lorsqu'une erreur est détectée, dans quel *ensemble de cycles* l'erreur s'est produite. L'avantage de cette

technique, est qu'au-delà d'une certaine valeur de *k*, il est moins coûteux en terme de surcoût matériel d'utiliser des points de sauvegarde, plutôt que de permettre une sauvegarde en continu. Toutefois comme le nombre *p* varie suivant la source considérée à l'intérieur d'un circuit, le nombre *k* à partir duquel créer des points de sauvegarde est avantageux varie d'un circuit à un autre.

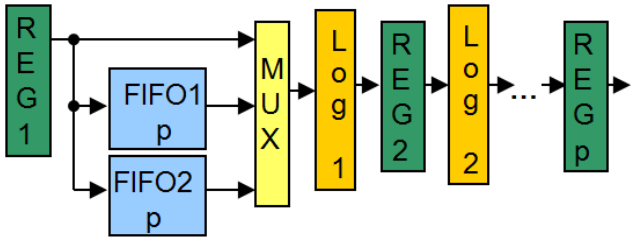

Fig. 4-12 : Sauvegarde des composants de stockage locaux avec point de sauvegarde

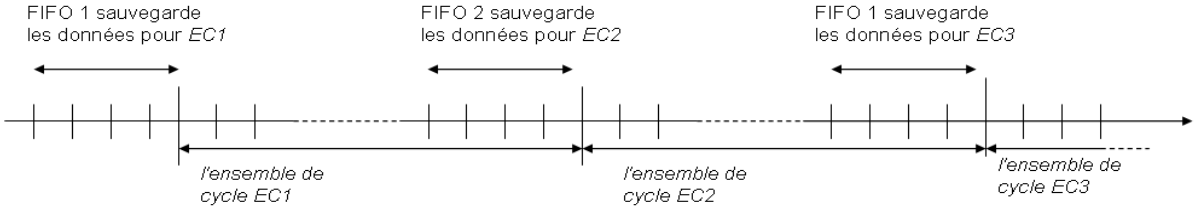

Fig. 4-13 : Chronogramme des ensembles de cycles.

4.4.2 Mécanisme de sauvegarde pour les mémoires

A la différence des FIFO associées aux registres, l'utilisation de points de sauvegarde ne permet pas de réduire le coût matériel au niveau des mémoires. En effet, on ne peut pas prédire quand les données vont être lues puis effacées durant chacun de ces *ensembles de cycles*. Aussi le système de CAM/FIFO doit sauvegarder les opérations durant tout un ensemble de cycles. Mais aussi comme pour les registres, si une erreur apparaît lors du dernier cycle d'un ensemble de cycles, il faut pouvoir revenir au début du cycle précédent. Il faut donc une FIFO/CAM sauvegardant $m+k$ opérations, $k-1$ étant le nombre de cycles avant l'activation du signal d'erreur, et m le nombre maximal de cycles entre deux points de sauvegarde (figure 4-14). En revanche la taille de la FIFO associée dépend de l'implémentation choisie pour restaurer les *p* étages du pipeline qui dépend de la mémoire.

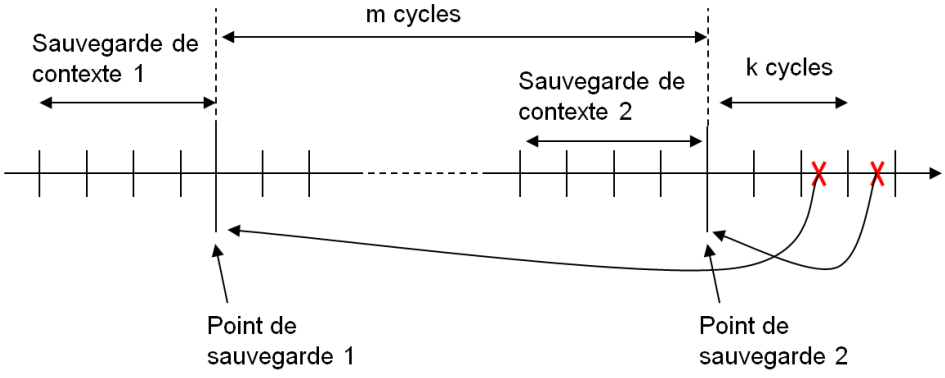

Fig. 4-14 : Chronogramme des EC et utilisation des points sauvegarde en fonction du signal d'erreur.

Dans une implémentation où ce sont les données lues qui sont sauvegardées dans la CAM (implémentation 1 et 2, figure 4-15), l'un des deux ensembles d'états pour la restauration d'un des EC est sauvegardé dans la CAM. En effet la sauvegarde de $m+k$ données lues, sauvegardées dans la CAM, comprend les p états liés au cycle le plus récent. Donc, dans le cas le plus défavorable, seulement les $p–1$ premiers états liés au cycle précédent doivent être préservés dans une FIFO supplémentaire (le dernier des p états faisant partie de l'ensemble des données sauvegardées dans la CAM/FIFO). En revanche lorsqu'il s'agit d'une implémentation où ce sont les données écrites qui sont sauvegardées dans la CAM (implémentation 3, figure 4-16), il faut conserver les deux FIFO de p étages relatives aux deux cycles.

Important : Pour éviter de commencer la sauvegarde de contexte du point de sauvegarde 3, avant d'être certain de ne plus avoir besoin du point de sauvegarde 1, la taille d'un ensemble de cycle m doit être supérieure à $k+p_M–1$. De cette façon, les k cycles suivants le point de sauvegarde 2, nécessaire pour être sûr que ce point de restauration n'est pas corrompu, auront été effectués au moment où débutera la sauvegarde de contexte pour la restauration du point de sauvegarde suivant.

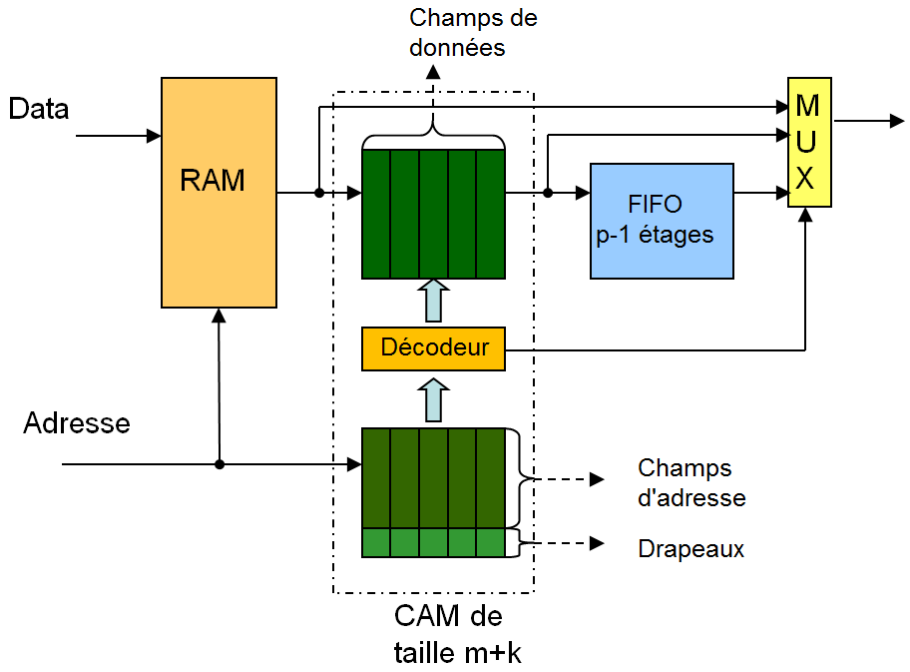

Fig. 4-15 : Implémentation 1 et 2 avec point de sauvegarde

4.5 Algorithme

D'un point de vue théorique, il n'y a pas d'ensemble à déterminer comme dans le chapitre 3 où les sources sont identifiées en fonction des chemins contaminables du circuit. En effet, ici les sources sont identifiées de proche en proche en identifiant les zones régulières du circuit.

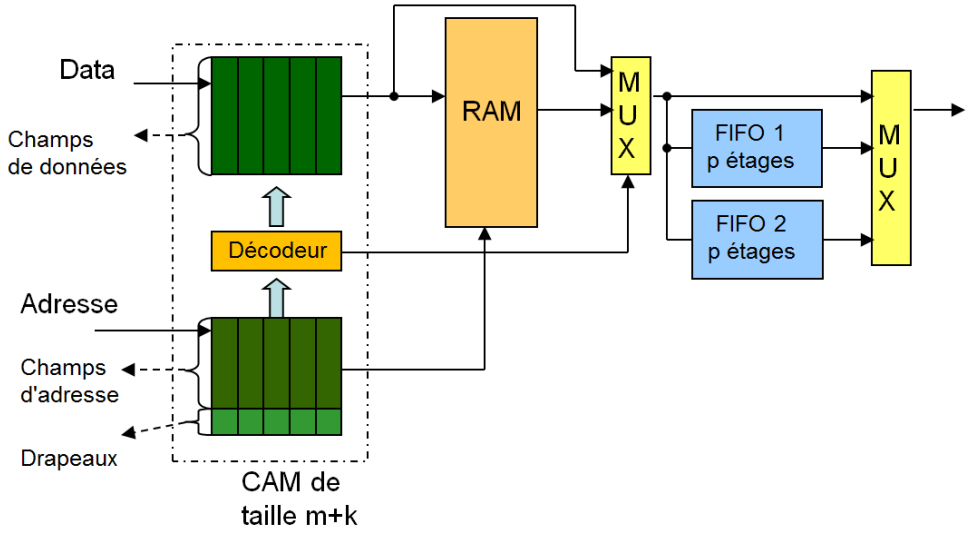

Fig. 4-16 : Implémentation 3 avec point de sauvegarde

Les zones régulières telles que montrées dans la figure 4-5 doivent vérifier les propriétés suivantes (ces propriétés ne sont pas vérifiées après l'étape d'optimisation).

Propriété 1 : chaque nœud d'une zone doit être à égale distance de toutes les sources dont il dépend. Cette condition tient à la définition donnée des zones régulières.

Propriété 2 : toutes les entrées d'un nœud, qui n'est pas un nœud source, doivent être situées à la même distance de leurs nœuds sources respectifs. Cette condition est là pour s'assurer que la restauration du contexte du nœud en question se déroule correctement. On pourrait imaginer que lorsque la **propriété 1** est vérifiée la **propriété 2** est automatiquement vérifiée : un contre exemple est donné dans la suite.

L'algorithme se décompose également en 4 étapes principales comme dans le chapitre 3 :

- L'algorithme identifie les pipelines réguliers, et du même coup les sources de chacune des parties régulières ;
- En fonction de l'implémentation choisie (avec ou sans point de sauvegarde), on en déduit la taille de chaque FIFO associée à une source.
- La fonction visant à réduire le coût matériel utilisée dans le chapitre 3 peut être réutilisée moyennant quelques modifications mineures.
- La fonction déterminant la machine à état rétablissant le bon contexte pour la réexécution des cycles durant lesquels l'erreur est survenue peut être réutilisée à l'identique, dans une implémentation sans point de sauvegarde, et avec quelques modifications mineures dans une implémentation avec point de sauvegarde.

4.5.1 Identification des sources et pipelines réguliers

Comme dans le chapitre 3 nous utilisons une représentation du circuit sous forme de graphe orienté. Pour déterminer les pipelines réguliers nous employons une variable de marquage triple appliquée à chaque nœud n du circuit $f(n)=(S_o(n), d_s(n), S_r(n))$:

- Lorsque $S_o(n)=1$ (source) le nœud est identifié comme étant un nœud source, $S_o(n)=0$ sinon.
- La valeur $d_s(n)$ est un entier qui indique dans le cas où $S_o(n)=0$ à quelle distance se trouve le nœud n du premier nœud source. On ne peut pas utiliser $d_s(n)=0$ pour identifier un nœud source, car c'est la valeur par défaut attribuée à chaque nœud au début de l'algorithme
- $S_r(n)$ est l'ensemble de nœuds sources desquels le nœud *n* dépend.

Compte tenu qu'il s'agit d'identifier des pipelines réguliers, tous les nœuds doivent vérifier à la fin de l'algorithme de marquage la **propriété 1**, se traduit par :

$$\forall n \in V, \exists! d \in N, d = d_S(n) / \forall s \in S_r(n), d_{min}(s,n) = d$$

De même, à la fin de l'algorithme, tous les nœuds vérifient également la **propriété 2** se traduit par :

$$\forall n \in V, S_O(n) = 0, \exists! d \in N, d_S(n) = d / \forall n' \in Ip(n), d_S(n') = d - 1$$

Le détail de l'algorithme est donné en Annexe F

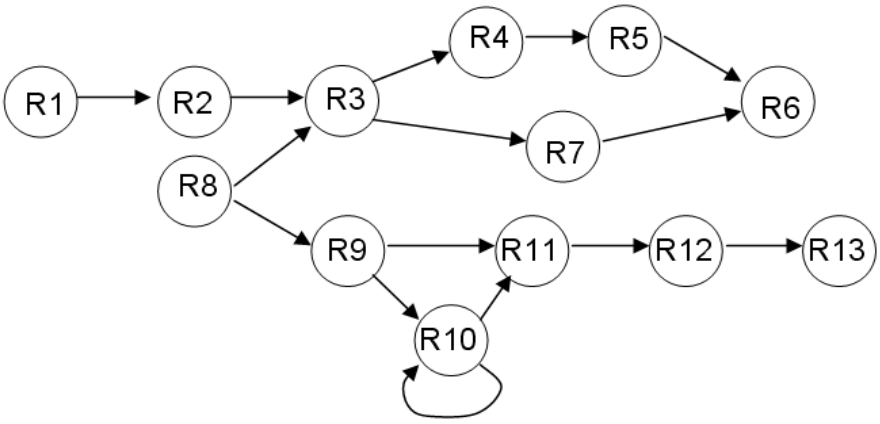

Fig. 4-17 : Graphe orienté du circuit de la figure 4-5

Le marquage du graphe orienté de la figure 4-17 se fait en plusieurs étapes. Au début de l'algorithme, les nœuds identifiés comme étant des nœuds sources sont des nœuds sans prédécesseurs. Ils correspondent soit au premier étage d'un port d'entrée du circuit, soit au port de sortie d'une mémoire. En effet pour les besoins de l'algorithme, on utilisera deux nœuds distincts pour représenter le port d'entrée et le port de sortie d'un composant de stockage de masse, et ces deux nœuds ne sont pas connectés entre eux. Ici *R1* et *R8* sont les deux seuls nœuds identifiés comme étant des nœuds sources au début de l'algorithme.

Tab. 4-1 : Marquage des nœuds de la figure 4-17, étape 1

n	R1	R2	R3	R4	R5	**R6**	**R7**	R8	R9	R10	R11	R12	R13
S_o	1	0	0	0	0	**0**	**0**	1	0	0	0	0	0
d_s	0	1	2	3	4	**5**	**3**	0	0	0	0	0	0
S_r	∅	R1	R1	R1	R1	**R1**	**R1**	∅	∅	∅	∅	∅	∅

Le tableau 4-1 montre l'état de l'algorithme de marquage appliqué au graphe 4-17 après avoir parcouru le chemin *(R1, R2, R3, R4, R5, R6)* puis après avoir atteint le nœud *R7* dans le chemin *R1, R2, R3, R4, R7*. A l'étape suivante, l'algorithme détectera un problème, car $d_s(R7)=3$, et donc à l'étape suivante on devrait avoir $d_s(R6)=4$. Or *R6* est déjà marqué avec $d_s(R6)=5$. Au cycle suivant *R6* sera donc marqué $S_o(R6)=1$. Mais comme *R8* a été marqué durant la phase d'initialisation $S_o(R8)=1$ (car *R8* n'a pas de prédécesseur), la source *R8* sera traitée avant *R6*. En traitant la source *R8*, on constatera un conflit similaire au moment de marquer *R3*. Ce qui conduira à marquer $S_o(R3)$, et à effacer tous les marquages associés aux successeurs de *R3*, jusqu'à *R6* non inclus, car $S_o(R6)=1$. Si la source *R6* avait déjà été traitée, les nœuds successeurs de *R6* n'auraient pas eu leurs marquages effacés (tableau 4-2).

Tab. 4-2 : Marquage des nœuds de la figure 4-17, étape 2

n	R1	R2	**R3**	**R4**	**R5**	R6	**R7**	R8	R9	R10	R11	R12	R13
S_o	1	0	**1**	**0**	**0**	1	**0**	1	0	0	0	0	0
d_s	0	1	**0**	**0**	**0**	0	**0**	0	0	0	0	0	0
S_r	∅	R1	**∅**	**∅**	**∅**	∅	**∅**	∅	∅	∅	∅	∅	∅

C'est ce qui arrive au moment de traiter le deuxième chemin issu de *R8* comme expliqué ensuite. On suppose dans le tableau 4-3 que le chemin *R8, R9, R11, R12, R13* a été traité avant le chemin *R8, R9, R10, R11, R12, R13*, et avant le chemin *R8, R9, R10, R10, R11, R12, R13*.

Tab. 4-3 : Marquage des nœuds de la figure 4-17, étape 3

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
S_o	1	0	1	0	0	1	0	1	0	0	0	0	0
d_s	0	1	0	0	0	0	0	0	1	0	2	3	4
S_r	∅	R1	∅	∅	∅	∅	∅	∅	R8	∅	R8	R8	R8

Au moment de traiter le chemin *R8, R9, R10, R11, R12, R13*, un conflit sera détecté au niveau du nœud *R11*, ce qui conduira au marquage $S(R11)=1$, et à effacer les marquages suivants

Tab. 4-4 : Marquage des nœuds de la figure 4-17, étape 4

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	**R10**	**R11**	**R12**	**R13**
S_o	1	0	1	0	0	1	0	1	0	**0**	**1**	**0**	**0**
d_s	0	1	0	0	0	0	0	0	1	**2**	**0**	**0**	**0**
S_r	∅	R1	∅	∅	∅	∅	∅	∅	R8	**R8**	∅	∅	∅

Puis l'algorithme détectera un autre problème avec le nœud R10 au moment de traiter le chemin *R8, R9, R10, R10*. On peut noter quel que soit l'ordre d'exploration des nœuds que le résultat final est le même, à savoir celui donnée par le tableau 4-5

Tab. 4-5 : Marquage des nœuds de la figure 4-17, étape finale

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
S_o	1	0	1	0	0	1	0	1	0	1	1	0	0
d_s	0	1	0	1	2	0	1	0	1	0	0	1	2
S_r	∅	R1	∅	R3	R3	∅	R3	∅	R8	∅	∅	R11	R11

Il y a toutefois une situation pour laquelle la procédure d'effacement du marquage doit être faite avec prudence, ce qui est illustré par le graphe de la figure 4-18

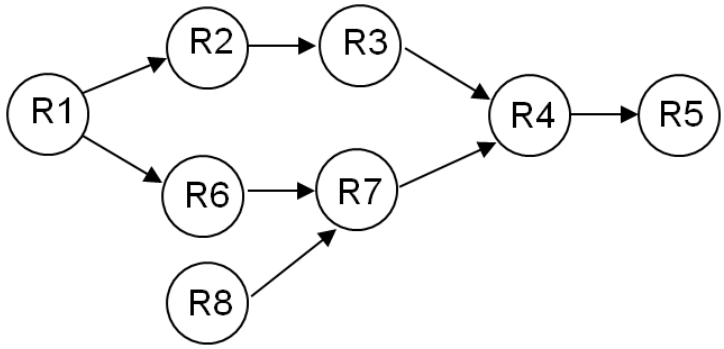

Fig. 4-18 : Exemple 2 pour l'algorithme de marquage

Après avoir traité la source *R1*, l'algorithme de marquage aura effectué les opérations résumées par le tableau 4-6. Au moment de traiter le nœud source *R8*, l'algorithme détectera un conflit avec le marquage de *R7*, vu que $d_s(R7)=2$, alors que l'algorithme souhaite marquer *R7* $d_s(R7)=1$. La procédure d'effacement souhaitera réinitialiser le marquage des nœuds *R7*, *R4* et *R5*, et *R7* sera marqué comme un nœud source. Mais dans ce cas, *R4* et *R5* seront uniquement associés à la source *R7*. Comme le marquage a effacé le lien de *R4* et *R5* avec *R1*, aucun autre conflit ne devrait être détecté. Seulement, au moment de la procédure de restauration du contexte, la FIFO de *R1* ne sera pas dimensionnée pour restaurer le contexte de *R4*. Par conséquent la restauration du contexte ne peut pas se faire. Dans cet exemple la **propriété 2** n'est pas vérifiée, alors que la **propriété 1** est vérifiée.

Tab. 4-6 : Marquage des nœuds de la figure 4-18, étape 1

n	R1	R2	R3	R4	R5	R6	R7	R8
S_o	1	0	0	0	0	0	0	1
d_s	0	1	2	3	4	1	2	0
S_r	∅	R1	R1	R1	R1	R1	R1	∅

Pour éviter cette situation, il faut détecter le problème au moment où on lance la procédure d'effacement. Au moment de traiter *R4*, on vérifie si ce composant a plusieurs entrées et, si oui, si le nœud qui le précède a déjà été marqué. Dans ce cas *R4* est marqué comme un nouveau nœud source, et la procédure d'effacement se poursuit. Le tableau 4-7 donne le résultat final de la procédure de marquage du graphe 4-18.

Tab. 4-7 : Marquage des nœuds de la figure 4-18, étape finale

n	R1	R2	R3	R4	R5	R6	R7	R8
S_o	1	0	0	1	0	0	1	1
d_s	0	1	2	0	1	1	0	0
S_r	∅	R1	R1	∅	R4	R1	∅	∅

4.5.2 FIFO additionnelle et cycle de décontamination

A l'issue de l'étape précédente l'ensemble des nœuds source n_s, marqués $S(n_s)=1$, est connue. De même la profondeur du pipeline régulier associé $p(n_s)=Max(d_s(n) / n_s \in Sr(n)) + 1$. En fonction de l'implémentation utilisée (sans ou avec point de sauvegarde), et du nombre k de cycles d'horloge avant le début de la phase de restauration d'état, on en déduit la taille de chaque FIFO telle qu'exprimée dans les paragraphes précédents. Ces FIFO sont donc dimensionnées pour pouvoir décontaminer les nœuds auxquels on les a associées. La fonction destinée à réduire le coût des FIFO additionnelles utilisée dans le chapitre 3 (et annexe D) peut être utilisée à l'identique. Simplement lorsqu'on utilise une solution à base de points de sauvegarde, il faut déplacer simultanément les deux FIFO associées à une source. Par conséquent la pénalité, dans ce type d'implémentation, pour utiliser un nouveau nœud source situé en entrée du nœud source initial est de deux étages de FIFO supplémentaires (un étage par FIFO). Lorsqu'il y a des FIFO associées à deux sources qui sont regroupées, elles sont automatiquement dimensionnées pour décontaminer les pipelines liés à chacune des sources. Le principe est en effet similaire à celui détaillé en début de chapitre par la figure 4-2 : à chaque fois que l'on déplace une FIFO vers l'arrière (en l'associant à un nœud situé en entré du nœud initial), la FIFO doit avoir un étage de plus afin de pouvoir restaurer l'état de ce nœud supplémentaire auquel on vient de l'associer.

Durant la phase de restauration d'état, les FIFO destinées à restaurer l'état à partir duquel la phase de réexécution va commencer doivent être débloquées (comme les différents étages des pipelines connectés) de façon précise. Tout comme dans le chapitre 3 le déblocage des FIFO se fait dans l'ordre inverse de leur taille, et le déblocage des registres est déterminé cycle après cycle avec les mêmes critères. Il s'agit en fait de la généralisation à l'ensemble du circuit de la méthode employée au chapitre 3 (et annexe E). Les FIFO étant dimensionnées en fonction du nombre d'étages de pipeline à restaurer, celles dont la taille est la plus importante doivent être activées en premier, car ce sont celles qui doivent restaurer le $k^{ème}$ état précédent d'un maximum d'étages. Ainsi que cela a été dit dans la section 4.3, cette restauration d'état dure p_M cycles d'horloge, p_M étant le plus grand nombre d'étages à décontaminer. En revanche, une fois que la phase de restauration de contexte est terminée, toutes les données restant dans les FIFO ne sont plus utilisées. Ceci était déjà prévu dans l'algorithme original, car lorsqu'un nœud source était décontaminé, la FIFO associée n'était plus utilisée. Or à la fin de la phase de restauration du contexte, tous les registres sont à nouveau débloqués, les données des FIFO ne sont donc automatiquement plus utilisées. Il n'y a pas besoin de machine à état particulière pour la phase de réexécution (hormis pour prendre en compte les données de la CAM, et pour sélectionner la bonne FIFO dans le cas d'une implémentation avec check-point durant les phases de sauvegarde). En effet l'ensemble du circuit fonctionne de manière normale durant les phases normales et les phases de réexécution.

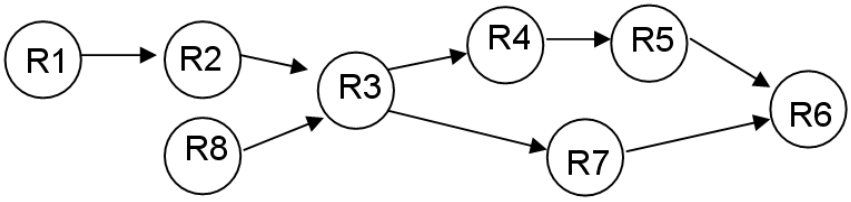

Fig. 4-19 : Exemple de graphe pour illustrer la phase de restauration d'état

Ces étapes sont illustrées par le fragment de graphes de la figure 4-19. Dans ce graphe, on suppose que l'on utilise la technique de retour de *k* cycles en arrière (le raisonnement serait identique avec une technique avec check point). Supposons par exemple que *k=1*. A la fin de la fonction d'optimisation, une FIFO de taille 6 sera associée à *R1*, et une FIFO de taille 5 sera associée à *R8*. Au premier cycle de la phase de décontamination, la FIFO de *R1* devient active et fourni l'état *R1*, six coups d'horloge avant le signal d'erreur. *R2* est débloqué, et le 5ème état de *R2* précédent l'activation du signal d'erreur est restauré. Au cycle suivant, *R3* est débloqué, la FIFO de *R8* devient active et fourni le 5ème état précédent le signal d'erreur de R8. Les 4èmes états précédents de *R2* et *R3* sont restaurés. Au troisième cycle, *R4* et *R7* sont débloqués et les 3èmes états précédents l'activation du signal d'erreur de *R2, R3, R4* et *R7* sont restaurés. Au quatrième cycle *R5* est débloqué et les 2èmes états précédents le signal d'erreur de *R2, R3, R4, R5* et *R7* sont restaurés. Au dernier cycle, *R6* est débloqué, les 1ers états précédents le signal d'erreur de *R2, R3, R4, R5, R6* et *R7* sont restaurés. Au cycle suivant la restauration du contexte est terminée, car les 1ers états précédents le signal d'erreur de *R1* et *R8* seront

fournis par leur FIFO respectives, permettant ainsi de ré-éxécuter le cycle durant lequel l'erreur est apparue.

4.6 Résultats expérimentaux

Comme pour le chapitre 3, les surcoûts des différentes implémentations envisagées ont été évalués sur les même quatre cas d'études. Le surcoût du système de CAM/FIFO n'est pas le facteur limitant de l'implémentation par rapport à la taille des composants de stockage de masse. De plus ce système a déjà fait l'objet d'une évaluation dans [TTR88][TT89] (voir figure 4-9). Les résultats se focalisent sur le surcoût par rapport à la puce (fonctions logique + composants de stockage isolés, comme dans le chapitre 3) des CAM et FIFO de sauvegardes additionnelles en fonction des implémentations proposées. Dans cette étude, on a également pris en compte le système de CAM/FIFO, car il est directement lié au type d'implémentation et d'algorithme choisi. Pour simplifier et harmoniser la notation, le modulateur OFDM avancé a été noté IP0. Les implémentations / algorithmes 1 et 2 ("Algo 1,2") ont été comparés à l'implémentation / algorithme 3 ("Algo 3"). De même ont été comparé la technique de retour de *k* cycles en arrière ("retour k cycles"), *k* étant le délai de détection d'une erreur, et la technique de point de sauvegarde ("check point"). On a également évalué la solution proposée par [TTR88] [TT89] qui donne un surcoût de +500% par composant de stockage isolé pour *k=4*.

Ce surcoût a été évalué en fonction du paramètre *k*, en considérant pour les implémentations avec point de sauvegarde la valeur minimale de *m* (*m=k+p–1)*, pour que l'implémentation des FIFO supplémentaires soit complétement optimisée. L'utilisation d'un paramètre *m* inférieur (pour réduire la taille de la CAM) conduirait à contraindre, lors des phases de marquage des nœuds et d'optimisation, la valeur maximale autorisée pour les paramètres *p.* En conséquence, plusieurs FIFO supplémentaires pour combler la différence doivent être introduite augmentant en général le surcoût, en utilisant une solution non optimisée. Ce surcoût n'est pas en général compensé par la réduction de la CAM. Il faudrait également considérer, en plus de ce surcoût, la pénalité dû à l'utilisation d'un système de détection de l'erreur transitoire. Ces techniques varient suivant les systèmes et les cas d'études considérés entre environ 30% et 170%, comme détaillé dans le chapitre 1. Ces surcoûts, au final très importants, doivent être inférieurs à une technique de TMR complète qui correspond à un surcoût de +200% pour l'ensemble de la puce. Les figures 4-20, 4-21, 4-22, et 4-23 représentent respectivement les surcoûts dus aux différentes techniques proposées, pour les cas d'études IP0, IP1, IP2 et IP3.

Pour le cas d'étude IP0, le surcoût relatif dû à l'utilisation de ces techniques est assez faible, notamment à cause de la faible proportion des composants de stockage isolés par rapport à l'ensemble de la puce. Ce faible ratio (moins de 20%), conduit à avoir même pour des valeurs élevées de *k* (supérieure à 10) une pénalité de surface inférieure à 100%. Les simulations confirment que l'implémentation/algorithme 3, le plus tolérant aux

fautes, est également le plus coûteux lorsqu'il s'agit d'effectuer un retour de *k* cycles d'horloge en arrière, *k* étant fixe. En revanche, cette même technique, mais appliquée en faisant des points de sauvegarde (check point), est à peine plus importante en surface que l'implémentation / algorithme 1 ou 2. Pour le cas de IP0, la technique la plus avantageuse d'un point de vue du coût matériel reste l'implémentation / algorithme 1, 2 permettant d'effectuer un retour de *k* cycles en arrière. Toutefois pour l'implémentation 3, et pour de grande valeurs de *k (k>16)*, l'implémentation avec check point est moins coûteuse qu'une implémentation avec un retour de *k* cycles en arrière. A l'inverse la technique provenant de l'état de l'art [TTR88][TT89], et qui permet une restauration de l'état précédent de *k* cycles en arrière en un cycle d'horloge a un surcoût quasiment de 50% supérieure à la technique la plus coûteuse que nous proposons, et plus de 100% supérieure à la technique de retour de *k* cycles en arrière. Ce coût doit d'ailleurs logiquement augmenter encore plus rapidement que les autres lorsque *k* augmente, même si nous n'avons pas fait ici les simulations. En effet, la technique employée est d'une complexité plus importante qu'une simple FIFO. De plus celle-ci est appliquée à tous les composants de stockage isolé. Le surcoût est nécessairement plus important qu'une technique où l'implémentation des FIFO a été étudiée pour minimiser le matériel supplémentaire. On peut remarquer que contrairement à une technique permettant un retour de *k* cycles en arrière, la progression du coût des techniques avec check point est linéaire. En effet à partir du moment où la taille de chaque FIFO est fixée (car elle ne dépend que du paramètre *p*, et pas de *k*), la taille du système CAM/FIFO est la seule à progresser avec *k*. Ceci n'est pas valable dans une implémentation prévue pour un retour de *k* cycles en arrière, car l'optimisation des FIFO dépend du paramètre *k*. Le coût ne devient linéaire qu'au-delà d'une certaine valeur de *k*, lorsqu'il n'y a plus de meilleure optimisation possible.

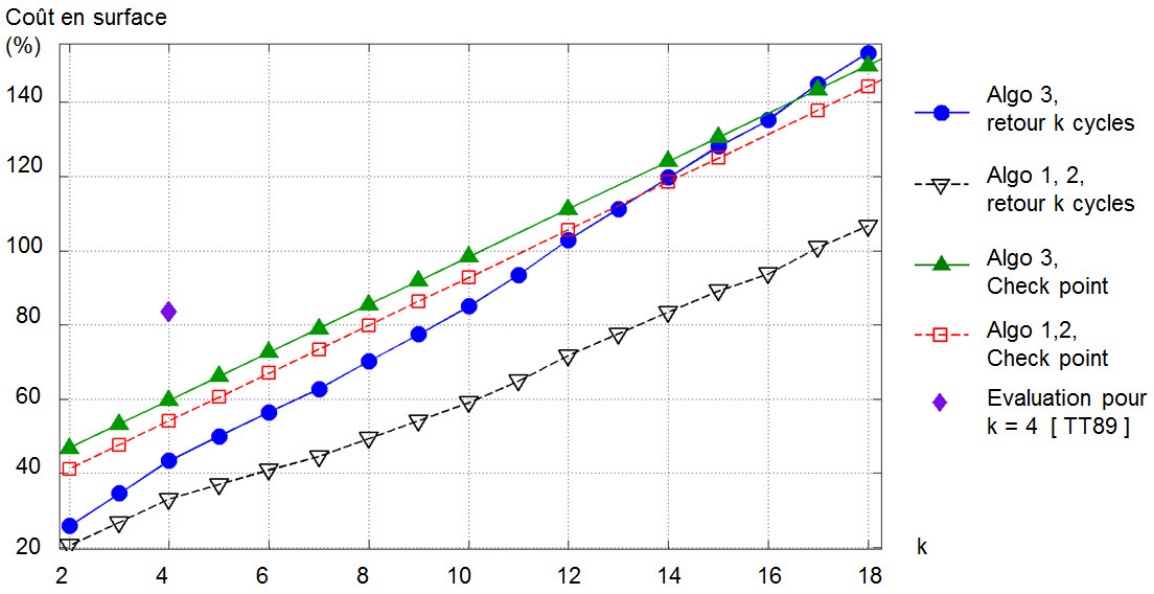

Fig. 4-20 : Coût en surface des systèmes de retour en arrière pour IP0

Les résultats pour le cas d'étude IP1 (Figure 4-21), sont plus défavorables dans le sens où dès que *k>4* les surcoûts sont supérieurs à 100%. De plus, l'implémentation avec check point est également rapidement plus défavorable qu'une implémentation pour un

retour de *k* cycles en arrière, quel que soit l'algorithme choisie. En réalité, comme pour IP0, l'implémentation 3 deviendrait moins coûteuse avec une implémentation avec check point pour des valeurs élevées de *k,* mais le surcoût en surface dépasserait alors les 300%. Notons que comparativement à [TT89], pour *k=4*, la moins bonne de nos solutions donne un surcoût de 175%, et la meilleure 90%, contre 190% pour l'état de l'art.

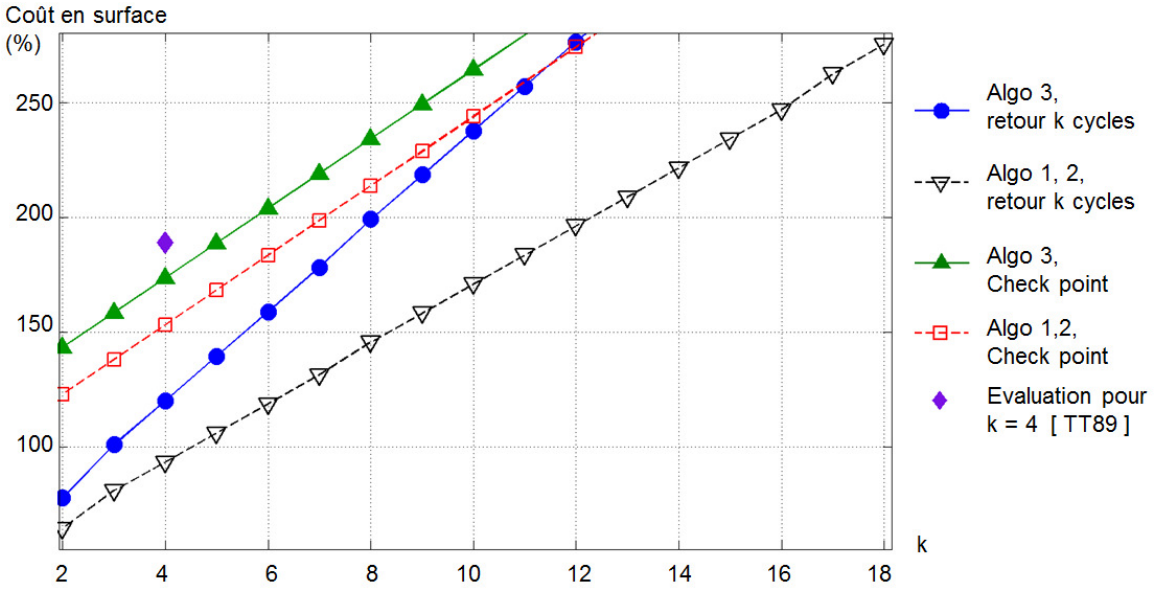

Fig. 4-21 : Coût en surface des systèmes de retour en arrière pour IP1

Les constatations sont très différentes pour IP2 (figure 4-22). Dans ce cas de figure, le nombre de port mémoire est relativement faible, alors que l'architecture est très irrégulière. Il en résulte un surcoût qui dépend beaucoup moins de la taille de la CAM, et plus des FIFO additionnelles. En conséquence, ce cas d'étude donne un exemple où l'implémentation avec check point est la moins coûteuse, quel que soit l'algorithme choisi pour la CAM, dès que *k>9*. Cette solution présente l'avantage, d'avoir des FIFO associées aux composants de stockage isolés dont les tailles sont indépendantes de *k*, contrairement à une solution avec un retour de *k* cycle d'horloge en arrière. Cette importance des composants de stockage isolés est confirmée par le surcoût énorme pour *k=4*, de la technique provenant de [TT89], de 260%, soit plus de 2,5 fois le surcoût de la technique de retour de *k* cycles en arrière.

Enfin pour IP3, figure 4-23, les courbes montrent un surcoût qui croit nettement plus rapidement, puisque quelle que soit la technique considérée, le matériel supplémentaire dépasse 200% dès que *k>10*. Si IP2 donnait un exemple de cas où l'implémentation avec point de sauvegarde devenait pour des valeurs élevée de *k* préférable dans tous les cas de figure, IP3 est de ce point de vue l'exemple contraire. En effet ici, l'implémentation avec un retour de *k* cycles en arrière reste préférable quel que soit l'algorithme choisie. Ceci est dû à un faible nombre de FIFO additionnelle pour les composants de stockage isolés par rapport au nombre de ports mémoire à protéger par une CAM. En effet pour une implémentation sans check point, la taille d'une CAM augmente en fonction de *k,* alors que pour une implémentation avec check point elle croit

avec *m+k,* soit en fonction de *2k.* La solution avec point de sauvegarde devient alors même plus coûteuse que la technique provenant de l'état de l'art, qui reste elle-même deux fois plus coûteuse que la technique de retour de *k* cycles en arrière que l'on propose.

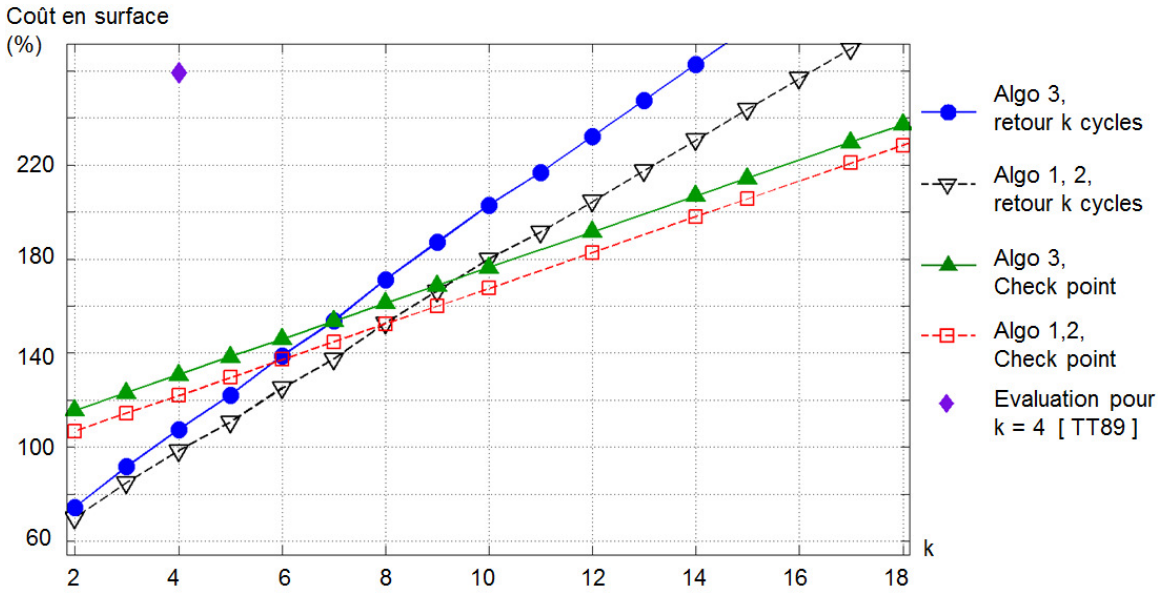

Fig. 4-22 : Coût en surface des systèmes de retour en arrière pour IP2

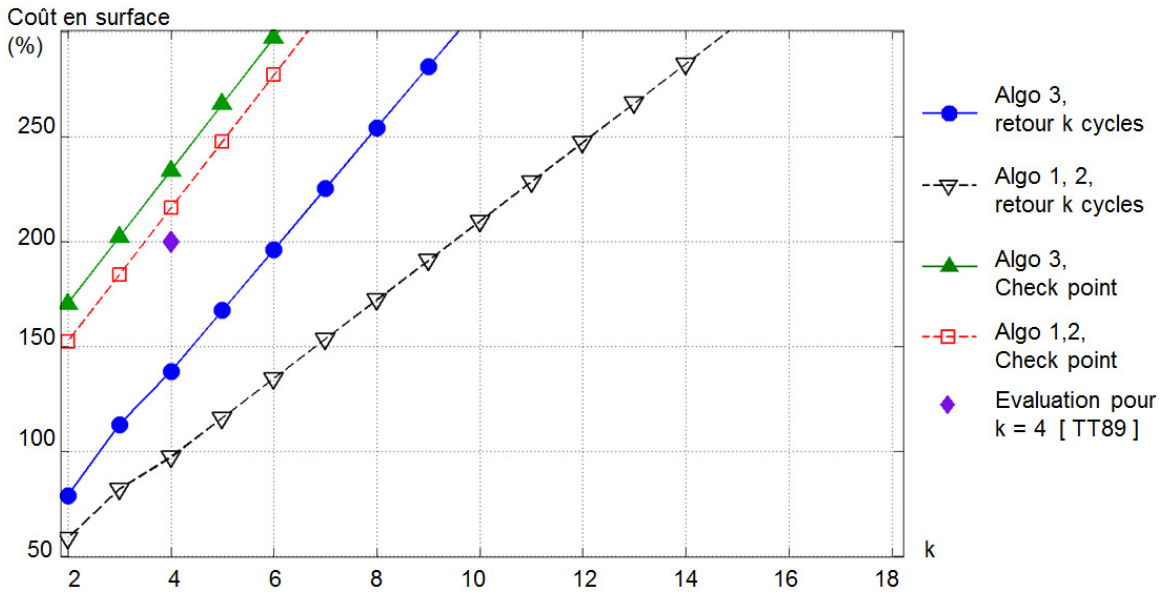

Fig. 4-23 : Coût en surface des systèmes de retour en arrière pour IP3

Les figures 4-24 et 4-25 représentent le gain apporté par l'utilisation de la fonction d'optimisation. Si dans le chapitre 3, l'apport d'une fonction d'optimisation pouvait suivant les cas d'étude être discutable, les courbes ci-dessous montrent que les résultats précédents auraient été bien plus coûteux sans l'utilisation de la fonction d'optimisation. Pour une implémentation avec un retour de *k* cycles d'horloge en arrière (figure 4-24), la fonction d'optimisation a permis une réduction du coût jusqu'à 60% pour IP0 et IP2, et 30% pour IP1 et IP3. La réduction est un peu moins importante pour l'implémentation / algorithme 3, à cause de la plus grande importance du système FIFO/CAM. De la même

manière la réduction du surcoût pour une implémentation avec check point (figure 4-25), diminue avec *k*, puisque seul la CAM change avec *k*, et que les FIFO sont optimisées une fois pour toute.

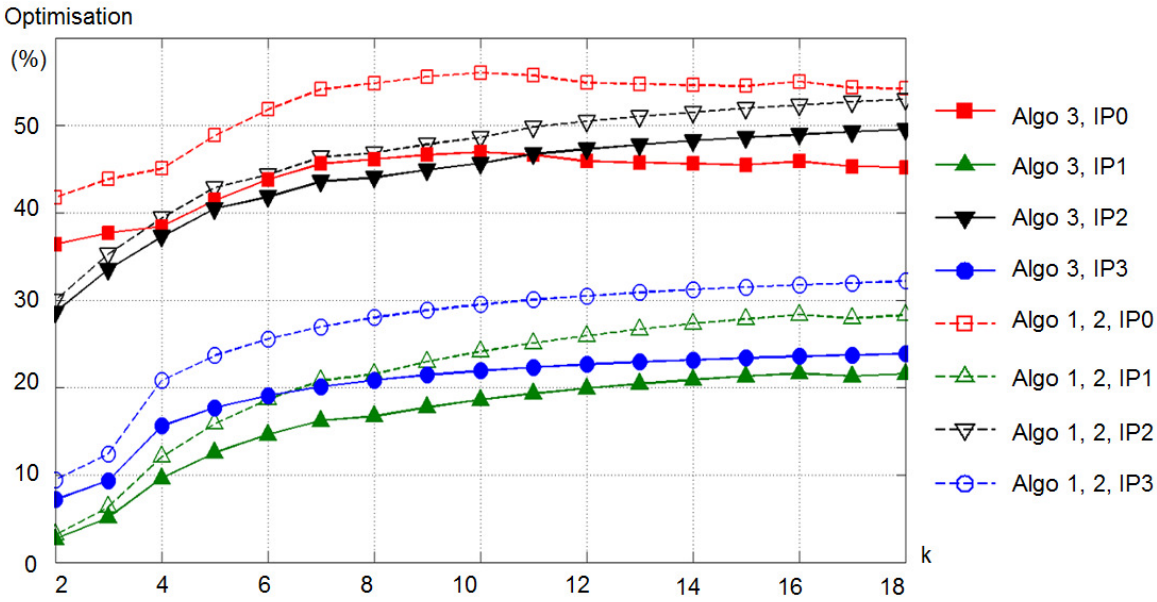

Fig. 4-24 : Réduction du coût grâce à la fonction d'optimisation pour les techniques de restauration de contexte en k cycles d'horloge

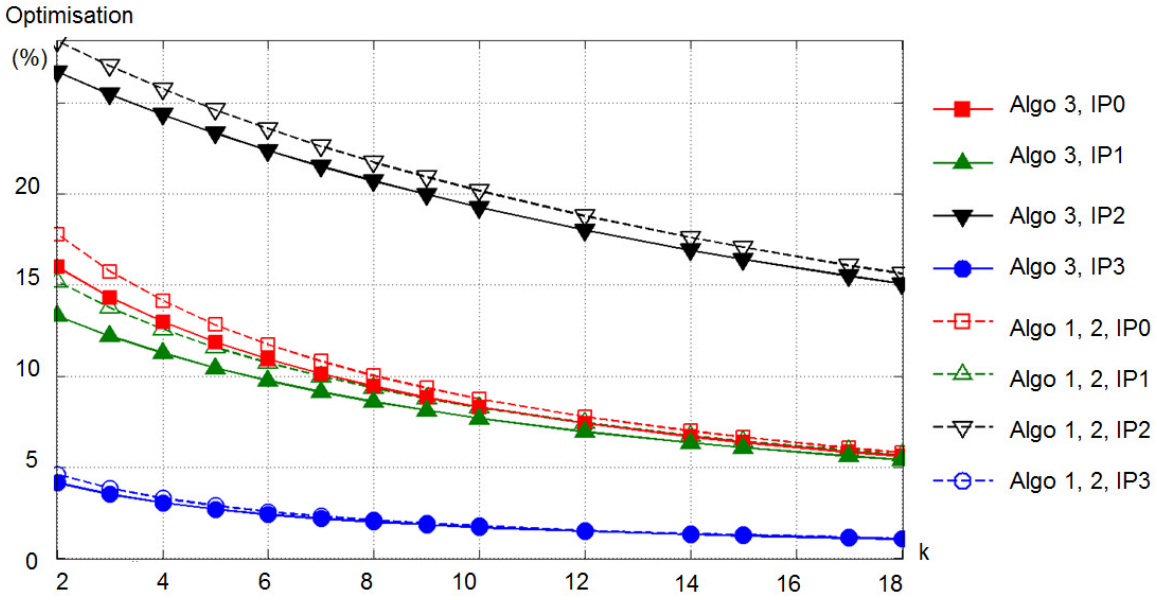

Fig. 4-25 : Réduction du coût grâce à la fonction d'optimisation pour les techniques de restauration de contexte avec point de sauvegarde

La contrepartie à cette optimisation est la pénalité de temps en termes de nombre de cycle d'horloge supplémentaire dû à la phase de restauration de contexte. Dans le cas le plus défavorable, qui correspond à la plus grande valeur du paramètre *p* relevé pour chacune des IP, cette perte est de 16 cycles pour IP0, 9 cycles pour IP1 et IP2, et 7 cycles pour IP3. Alors qu'à l'inverse la technique proposée par [TTR88][TT89] concède un seul

cycle d'horloge pour restaurer le $k^{ème}$ état précédent, mais avec un surcoût en surface très important, ainsi que nous l'avons démontré. Reste les *k* cycle à ré-exécuter, qui eux dépendent du système de détection utilisé. Dans le cas d'une implémentation avec point de sauvegarde, ces *k* cycles à exécuter peuvent dans le cas le plus défavorable être augmentés de *m* cycles, si l'erreur se produit lors du dernier cycle appartenant à un ensemble de cycles.

4.7 Surcoût global des techniques proposées

Dans ce paragraphe nous faisons le bilan global de l'implémentation des techniques proposées dans ces chapitres. Ce surcoût matériel est calculé par rapport à l'ensemble de la puce, en prenant en compte le surcoût par rapport aux blocks mémoire de la technique d'utilisation des codes correcteurs sans pénalité de temps, ainsi que le surcoût des différentes techniques de restauration de contexte proposées. Ce bilan ne tient donc pas compte de l'implémentation des codes correcteurs, et de la technique de détection d'erreur utilisée, dont le coût est très variable comme exposé dans le chapitre 1. Ces résultats sont exposés dans le tableau 4-8. La colonne *Roll-back* est relative à la fois l'implémentation de la technique de restauration de contexte proposée, et le nombre de cycle *k* précédent la détection de l'erreur et l'activation de la phase de restauration de contexte. Les colonnes *SEC* désigne une implémentation où la mémoire a été protégée par un code de Hamming, et *TEC* une mémoire protégée par un code BCH triple correcteur. Le surcoût dû au code BCH par rapport au code de Hamming implique que l'utilisation de notre technique pour utiliser les codes correcteurs sans pénalité de temps est relativement moins coûteuse pour le code BCH triple correcteur, comme les résultats l'ont montré au chapitre 2.

Globalement, on constate que les résultats sont en majorité dominés par le surcoût dû à l'utilisation des codes correcteurs sans pénalité de temps. Ceci s'explique par le fait que la surface des circuits étudiés est largement constituée de mémoire (96% pour IP0, 81% pour IP1, 91% pour IP2 et 97% pour IP3). Ces pourcentages sont ceux des mémoires non protégées, et l'ajout d'un code correcteur accroît encore leurs importances. Cela justifie des surcoûts relativement faibles par rapport à ceux exposés dans le paragraphe précédent, excepté dans le cas de IP1. Un autre paramètre important est la taille des blocks mémoires, puisque par rapport aux résultats présentés au chapitre 2 le surcoût relatif dû à l'utilisation de code correcteur sans pénalité de temps est d'autant plus faible que la mémoire est importante. Contrairement au chapitre 2, on a considéré ici l'ensemble des blocks mémoires pour IP0 soit 3,6Mbits, ce qui en fait le circuit ayant la mémoire la plus importante de nos quatre cas d'études. A l'inverse IP1 n'utilise que 24kbits de mémoire, et les circuits IP2 et IP3 respectivement 821kbits et 184kbits de mémoire embarquée. Ceci, ajouté au fait que la mémoire ne représente que 81% de la puce de IP1, implique que ce cas d'étude est le seul des quatre cas où le surcoût global est dominé par la technique de restauration de contexte. Au final on constate que, suivant les différents paramètres, ces surcoûts varient de 2,4% à 6% pour IP0, de 23% à 63% pour

IP1, de 5% à 16% pour IP2, et de 8,4% à 19% pour IP3. A noter que ces surcoûts ont été évalués en considérant de manière séparée les ressources additionnelles pour la restauration de contexte et les ressources pour l'utilisation des codes correcteurs sans pénalité de temps. Il est possible, pour réduire ce surcoût, de mettre en commun ces différentes ressources, mais au prix d'un temps de décontamination plus important dans le cas d'une erreur signalée par un code correcteur, les ressources pour la restauration de contexte n'étant pas forcément situé aux mêmes endroits du circuit. Cela représenterait toutefois une réduction du matériel supplémentaire qui serait assez faible.

Tab. 4-8 : Surcoût global des techniques de restauration de contexte et d'utilisation des codes correcteurs sans pénalité de temps

Roll-back		IP0		IP1		IP2		IP3	
Technique	k	SEC	TEC	SEC	TEC	SEC	TEC	SEC	TEC
k cycle roll-back Algo 1	3	+3,0%	+2,4%	+29%	+23%	+6,8%	+5,0%	+10%	+8,4%
	6	+3,5%	+2,8%	+36%	+28%	+9,2%	+6,7%	+11%	+9,0%
	9	+4,0%	+3,1%	+42%	+32%	+12%	+8,4%	+12%	+9,7%
	12	+4,5%	+3,4%	+48%	+36%	+14%	+10%	+14%	+10%
k cycle roll-back Algo 2	3	+3,3%	+2,7%	+33%	+26%	+7,1%	+5,2%	+11%	+8,8%
	6	+4,1%	+3,2%	+41%	+31%	+10%	+7,3%	+13%	+9,9%
	9	+4,8%	+3,7%	+51%	+38%	+13%	+9,4%	+14%	+11%
	12	+5,6%	+4,1%	+59%	+43%	+16%	+11%	+16%	+12%
Check point Algo 1	3	+3,7%	+2,9%	+39%	+30%	+8,6%	+6,2%	+12%	+9,6%
	6	+4,3%	+3,3%	+42%	+32%	+10%	+7,3%	+14%	+11%
	9	+5,0%	+3,8%	+51%	+38%	+11%	+8,1%	+16%	+12%
	12	+5,7%	+4,2%	+59%	+44%	+13%	+9,0%	+18%	+13%
Check point Algo 2	3	+4,0%	+3,1%	+42%	+32%	+9,2%	+6,7%	+13%	+9,9%
	6	+4,7%	+3,5%	+48%	+36%	+11%	+7,7%	+15%	+11%
	9	+5,3%	+4,0%	+55%	+41%	+12%	+8,4%	+17%	+12%
	12	+6,0%	+4,4%	+63%	+47%	+14%	+9,4%	+19%	+13%

4.8 Conclusion

Nous avons présenté dans ce chapitre, la généralisation du principe développé dans les chapitres précédents de la technique de restauration d'état. Notre méthode fourni un coût matériel très inférieure aux autres techniques du même type. En contrepartie, la restauration du contexte nécessite quelques cycles supplémentaires. Plusieurs techniques ont été proposées : restauration du contexte *k* cycles d'horloge précédent l'activation du signal de détection, ou restauration du contexte à des instants prédéfinis. Les cas de figures où une implémentation avec check point est moins coûteuse qu'une technique de retour de *k* cycles en arrière sont limités par la taille de la CAM. Cette dernière constitue souvent le facteur limitant pour l'utilisation de notre technique, et une implémentation avec check point n'est alors intéressante que dans le cas d'un circuit complexe comportant un faible nombre de port mémoire. La technique de protection des mémoires la plus tolérante aux fautes, provenant de l'état de l'art permet de gérer les erreurs d'adressage,

mais coûte beaucoup plus chère que les autres implémentations proposées. L'intérêt de ces techniques dépend toutefois étroitement du système de détection d'erreurs utilisé. En effet son surcoût matériel n'est pas inclus dans les résultats fournis, et le surcoût de la solution utilisée dépend du temps de détection. Ce temps de détection de même que le coût de la solution dépend également de l'architecture que l'on souhaite protéger. Comparativement toutefois aux autres techniques existantes, la technique de retour de *k* cycles en arrière proposée permet d'effectuer une restauration d'états à moindre coût, souvent deux fois moins que les autres techniques. Pour les composants de stockage isolé, la technique architecturale provenant de l'état de l'art donnait un surcoût de 500% par composant de stockage. Ici en mettant en commun les ressources pour plusieurs composant de stockage le coût matériel est nettement réduis, même si celui-ci dépend de l'architecture du circuit, jusqu'à plus 2,5 fois inférieur par rapport à l'état de l'art. De plus, la technique de restauration de contexte *k* cycles d'horloge supplémentaire est en général moins coûteuse en matériel qu'une solution avec check point, et également permet une restauration de contexte plus rapide. Nous avons également évaluer le surcoût global des deux techniques proposées dans le chapitre 2 et dans le chapitre 4 en considérant l'ensemble des circuits servant de cas d'étude, avec leur mémoire embarquées. Les résultats sont très variables en fonction des circuits considérés. Mais excepté dans un cas où la mémoire embarquée représente une plus faible partie de la puce (81%), ce qui conduit à des coûts globaux au-delà des 30%, les résultats donnent des surcoûts faibles en dessous des 20%, et souvent inférieurs à 10%. La grande place des mémoires embarquées dans les circuits modernes font qu'au final le surcoût dû à la technique de restauration de contexte est relativement faible par rapport à l'ensemble de la puce.

Conclusion générale

L'évolution technologique de la microélectronique a conduit à une perte significative de la fiabilité des circuits due, pour une partie non négligeable, à une sensibilité accrue à des phénomènes extérieurs aux circuits. Les neutrons énergétiques et les particules alpha crées par l'interaction des rayonnements cosmiques avec l'atmosphère terrestre constituent actuellement la source extérieur d'erreur la plus importante. Cela se caractérise par une augmentation importante des erreurs transitoires. L'objectif de ces travaux était de proposer de nouvelles techniques de protection qui puissent être à la fois efficaces, facilement intégrables et peu coûteuses en matériel supplémentaire.

Les études récentes sur la fiabilité des circuits montrent que les mémoires RAM sont les premières concernées par ces phénomènes d'erreurs transitoires. Certes leurs sensibilités par cellule a tendance à diminuer, mais l'augmentation de leur densité maintien leur taux de SEU dans le cas des mémoires DRAM, et l'augmente dans le cas des mémoires SRAM. Cela conduit au final à un accroissement du taux erreurs multiples et de l'étendu de ces erreurs. L'augmentation de la multiplicité des erreurs limite l'efficacité du multiplexage de colonne, qui de plus n'est pas compatible avec les mémoires récentes, résistantes aux variations de processus de fabrication. Les codes correcteurs d'erreurs sont l'une des solutions les plus utilisées aujourd'hui car ils permettent d'assurer une bonne fiabilité à un coût modéré. Mais la nécessité de protéger les mémoires embarquées contre les erreurs multiples conduit à utiliser des codes plus complexes qu'avant. Dans le même temps, ces codes ont une influence négative sur la bande passante du système. Les circuits de codage et de décodage induisent en effet une pénalité de temps qui peut être inacceptable dans un certain nombre de circuits.

Nous avons proposé dans ce mémoire une méthode universelle pour les architectures synchrones pour résoudre ce problème. En faisant subir un traitement différent aux bits de données et aux bits de contrôle, la pénalité de temps dû à l'utilisation d'un code correcteur lors de l'écriture d'une donnée en mémoire est supprimée. De même, en transmettant les données aux modules de calcul avant de détecter une erreur, on supprime la pénalité de temps lors de la lecture d'une donnée en mémoire. Il ne reste que quelques cycles d'horloge supplémentaires pour effectuer la décontamination quand une donnée erronée est propagée dans le circuit avant que l'erreur ne soit détectée. Cette pénalité représente une perte de performance insignifiante et est négligeable. La contrepartie est de devoir ajouter des FIFO et des multiplexeurs supplémentaires pour restaurer le contexte du circuit au moment où la donnée erronée a été lue afin que la décontamination du circuit se fasse correctement. Cette technique est facile à insérer pour une mémoire connectée à une architecture régulière, et nécessite une étude plus approfondie pour être utilisée dans les architecture les plus complexes. Elle a été testée dans une architecture de modulateur OFDM avancé avec divers codes correcteurs. Les simulations temporelles ont démontré son efficacité, avec un coût matériel et énergétique relativement faible. Les expérimentations donnent un surcoût en matériel de 3% et

environ 6% de consommation électrique supplémentaire. Cependant, en retour, la fréquence originale du circuit est maintenue, sans détérioration de la bande passante en l'absence d'erreurs.

Toutefois l'implémentation de cette solution nécessite une étude détaillée de l'architecture de chaque circuit qui peut être difficile dans certains cas. L'implémentation pour la partie "écriture en mémoire" de la solution ne dépend pas de l'architecture considérée. Elle ne présente donc pas de difficulté particulière. A l'inverse, l'implémentation de la solution pour la partie "lecture en mémoire" dépend de l'architecture et du nombre de cycles d'horloge *k* avant détection de l'erreur. Toutefois une connaissance très détaillée des opérations effectuées par le circuit n'est pas indispensable. Pour représenter un circuit nous avons utilisé un graphe orienté, dont les nœuds représentent les composants de stockage, et les arcs leurs interconnexions, en même temps que le sens de propagation des données.

A partir de cette représentation il a été possible de déterminer les FIFO à ajouter pour implémenter la solution de manière automatique. D'une description formelle des nœuds à protéger, et de la taille des FIFO à ajouter, ont été conçus les différents algorithmes analysant le circuit visé et déduisant matériel supplémentaire. Contre la perte de quelques cycles d'horloge supplémentaires lors de la correction d'une donnée erronée, en plus de ceux déjà perdus en décontaminant le circuit, il est également possible de changer l'emplacement de ces ressources matérielles supplémentaires afin d'en réduire l'importance. Ce délai supplémentaire est toutefois toujours négligeable si on tient compte de la fréquence d'apparition d'une erreur. L'optimisation est basée sur le principe de pouvoir réduire le nombre de lieux d'implémentation des ressources en recherchant des prédécesseurs communs aux composants de stockage auxquels une FIFO a été attribuée. A l'inverse, si le temps de décontamination doit être minimisé, il est possible de conserver ces ressources au plus près des zones à décontaminer. Les cycles de décontamination sont déterminés avec précision dans une dernière étape ce qui permet d'en déduire la FSM pour la phase de correction de l'erreur. Notre algorithme, a été testé sur quatre cas d'études aux architectures variés. Les tests ont donnés des résultats variables en fonction de leurs irrégularités. Pour les architectures les plus régulières, les ressources placées au plus près des zones à décontaminer constituent la meilleure solution possible. Pour les architectures plus complexes, mis à part pour de faibles valeurs de *k*, les possibilités d'optimisation peuvent être importante et conduire à une réduction de 20% des FIFO à ajouter. Au final le coût matériel à payer varie entre 2% et 11% du circuit (sans les mémoires). Cela permet de restaurer l'état d'une zone contaminée d'une puce électronique lorsqu'une donnée erronée a été lue en mémoire.

Dans un dernier chapitre nous avons généralisé ces principes à la correction d'une erreur quelconque dans un circuit, et à la restauration de l'état complet du circuit. On part du principe que le circuit est doté d'un système de détection permettant de repérer l'apparition d'une erreur, où qu'elle soit dans le circuit. La restauration de l'état du circuit tel qu'il était le cycle précédent l'apparition de l'erreur transitoire, devrait permettre

d'effacer l'erreur en ré-exécutant le cycle suivant. Au final plusieurs techniques ont été proposées : la restauration du contexte *k* cycles d'horloge précédent l'activation du signal de détection, ou la restauration du contexte à des instants prédéfinis. Cette dernière méthode ne présente d'intérêt que dans le cas d'un circuit complexe ayant un faible nombre d'accès en mémoire. De même, la technique de sauvegarde de contexte des mémoires la plus tolérante aux fautes est très coûteuse à utiliser. Cette technique reporte l'écriture en mémoire des données, mais permet de gérer les erreurs d'adressage. Cette technique coûte beaucoup plus chère lorsqu'elle est employée pour restaurer le contexte *k* cycles d'horloge en arrière par rapport à une technique ne garantissant pas la protection contre les erreurs d'adressage. D'une manière générale la technique de restauration de contexte un nombre *k* fixé de cycles d'horloge en arrière est souvent la moins coûteuse en matériel par rapport à l'état de l'art, et moins coûteuse en matériel et en cycle d'horloge supplémentaire par rapport à une technique avec points de sauvegarde.

L'intérêt des diverses solutions proposées dépend toutefois étroitement du système de détection d'erreurs utilisé, ainsi que de l'architecture des circuits à protéger. Toutefois, comparativement aux autres techniques existantes, celles-ci permettent d'effectuer une restauration d'états à moindre coût. En mettant en commun les ressources pour restaurer les états de plusieurs composant de stockage le coût matériel est nettement réduis, jusqu'à 2,5 fois moins par rapport à l'état de l'art. La contrepartie, comme pour la correction des erreurs provenant de la mémoire, est de devoir dépenser quelques cycles d'horloge en plus pour effectuer la correction. Une partie non négligeable de ces cycles supplémentaires est dédiée à la restauration d'un contexte correct ayant précédé l'apparition de l'erreur, et à partir duquel le circuit reprend son fonctionnement normal. Selon les architectures testées, ce nombre varie entre 7 et 16 cycles d'horloge. La perte engendrée reste toutefois insignifiante.

On peut noter que ces deux techniques, la protection des mémoires avec un code correcteur et la restauration d'état de l'ensemble du circuit, sont compatibles. En effet, le matériel supplémentaire employé dans la technique de restauration de contexte peut être utilisé pour décontaminer un circuit après avoir lu une donnée corrompue en mémoire. Simplement dans le cas le plus général, le nombre de cycle pour effectuer la décontamination est plus important. L'explication tient au fait que les ressources matérielles ne sont pas en général situées aux mêmes endroits que dans le cas où seulement la mémoire est protégée. La restauration des données nécessaires pour effectuer la décontamination prendrait alors quelques cycles supplémentaires. Sans pour autant mettre en commun les ressources de ces deux techniques, leur implémentation commune conduit à un surcoût global assez faible à cause de l'importante place prise par les mémoires. Aussi par rapport à des circuits, dont les mémoires représentent plus de 90% de la puce, l'ensemble de ces deux techniques donnent des surcoûts en dessous de 20%, et le plus souvent compris entre 5 et 15%.

Une implémentation matérielle de cette technique de restauration d'état avec un système de détection d'erreur dans un circuit réel permettrait de confirmer l'avantage de la

technique proposée dans le dernier chapitre. De manière plus générale, la méthode d'insertion automatique proposée dans les chapitres 3 et 4 doit être incorporée dans le flot de conception afin de confirmer sa faisabilité et sa capacité à minimiser le surcoût matériel. Cela passe par une poursuite du développement du code écrit, afin d'une part d'être intégré à un outil de conception, et d'autre part d'être éventuellement retravaillé pour limiter sa complexité et éviter un impact trop important sur le temps de développement des circuits. Avec le recul, et en particulier pour la technique de restauration de contexte sur l'ensemble d'un circuit, une étude sur l'impact d'une particule sur un circuit ainsi protégé pourrait s'avérer nécessaire. En particulier une particule peut-elle provoquer un SET sur les fonctions combinatoires, et en même temps provoquer un changement d'état sur les données sauvegardées précédemment dans les FIFO additionnelles ? En ayant pris parti de réduire au maximum le nombre de composant associé à une FIFO, cela réduit la probabilité d'un tel événement. C'est la raison pour laquelle ces FIFO supplémentaires n'ont pas été protégées par un système de bit de parité comme dans l'état de l'art. Avec la réduction des dimensions, cette question doit toutefois être soulevée, et vérifiée par une expérimentation matérielle.

ANNEXES

Annexe A. Justification de la convention $d_{min}(LM,v)$

Dans la section 3.2. la convention suivante $d_{min}(LM,v)=min\{d_{min}(lm,v),lm\in LM\}$, avait admise sans plus d'explication quant à son utilisation. Il avait été alors souligné que dans la figure 3-3 que $SC'_3=\{b, i\}$. On pouvait alors remarquer que le nœud *b* était considéré comme une source contaminable alors qu'en toute logique, *b* ne peut être contaminé que lorsque l'erreur provient de *lm2*. Alors, dans ce cas de figure, *b* n'est pas une source. A l'inverse lorsque l'erreur provient de *lm1, b* est une source extérieure non contaminée. Ceci est un des effets de l'utilisation de la convention $d_{min}(LM,v)=min\{d_{min}(lm,v), lm\in LM\}$, et qui ne permet pas de faire la distinction entre les chemins contaminés par l'une ou l'autre des mémoires. Cependant, le fait de ne pas faire la distinction entre les cas n'influe pas sur le dimensionnement de la FIFO associée à la source. En effet la taille des FIFO ajoutées est déterminée en fonction de la distance qui sépare le nœud à décontaminer de la mémoire. Par conséquent, que *b* soit considéré comme une source contaminable ou non, la FIFO associée possède le même nombre d'étages : $k–d_{min}(LM, c)+1$. Or dans le calcul de $d_{min}(LM, c)$, le fait d'utiliser la convention n'a pas d'influence sur le résultat final qui ne retiendrait que le cas le plus défavorable. La seule chose qui change c'est, dans le cas où l'on considère *b* comme une source contaminable, que la FIFO n'est pas utilisée après que *b* soit décontaminé. Ceci implique de décontaminer *b* même si en réalité *b* n'avait pas été traversé par une donnée contaminée.

Cependant, supposons maintenant que le fragment de circuit représenté par ce graphe soit un circuit ayant une mémoire multiport, *lm1* et *lm2* étant deux de ces ports. Supposons également que durant un cycle les deux ports mémoire puissent lire la même donnée. Dans ce troisième cas de figure, *b* est bien une source contaminable. C'est bien à ce dernier cas de figure, le plus général, qu'il faut se référer. Ceci justifie donc a posteriori l'emploi de la convention deux fois citée. La distinction entre les différents cas de figure, si celle-ci est souhaitable, peut être faite au niveau de la machine à état et des cycles de décontamination, avec *l'algorithme 6* (Annexe E). Deux solutions sont possibles. On peut ne faire qu'une seule machine à état qui effectuera systématiquement une décontamination complète, sans tenir compte du port mémoire d'où provient l'erreur. L'inconvénient de cette solution et une consommation accrue pendant les cycles de décontamination, puisque toutes les FIFO de toutes les sources sont utilisées, y compris pour décontaminer des chemins non contaminés. L'autre solution est d'appliquer l'algorithme déterminant les cycles de décontamination à chaque port mémoire. La FSM ainsi déterminée fera ensuite la décontamination en fonction du (ou des) signal d'erreurs reçu.

Annexe B. Exploration des chemins contaminés

Cette annexe concerne l'algorithme présenté dans la section 3.3.2 permettant de déterminer tous les chemins contaminables par une donnée lue en mémoire comportant une faute. La détection, se fait en *k* coups d'horloge. Un parcours en profondeur explore les chemins d'un graphe un par un. A chaque étape, *l'algorithme 1* explore le premier nœud connecté en sortie du nœud actuel, et ainsi de suite, de proche en proche, jusqu'à atteindre un nœud n'ayant plus aucun nœud connecté en sortie (ou que toutes les sorties aient été explorées). L'algorithme revient alors au dernier nœud exploré dont tous les nœuds connectés en sortie n'ont pas été visités.

Ici la "profondeur" de l'exploration est limitée par le nombre *k*. Donc, si on considère le circuit représenté sous la forme d'un graphe orienté, on considère tous les chemins de longueur *k* partant des nœuds représentant les ports de sortie des mémoires. On limite donc le nombre de nœuds traversés, contrairement à une exploration en profondeur classique. En revanche, afin d'éviter que l'exploration ne prenne un temps infini, un parcours en profondeur classique n'explore jamais deux fois le même nœud pour un même chemin. En effet, si le graphe comportait des boucles, cela conduirait à les parcourir de manière illimitée. Ici, puisque la longueur des chemins est limitée, cette restriction ne s'applique pas, et chacun des chemins obtenus peut contenir plusieurs fois le même nœud.

Notations particulières :

CP(p, i) : il s'agit du $i^{ème}$ composant du chemin numéro *p*. Ce composant est atteint en *i-1* cycles d'horloge.

LIFO(i) : La pile contenant tous les composants connectés en sortie du $i\text{-}1^{ème}$ composant du chemin exploré actuellement.

Algorithme 1 : détermination des chemins CPk

```
1:  p=0
2:  Pour chaque nœud lm∈LM faire
3:      p ← p+1
4:      i ← 2
5:      CP(p,1) ← lm
6:      Tant que i<k+1 faire
            /*Exploration du nœud suivant*/
7:          LIFO(i) ← Op(CP(p,i-1))
8:          Si Card(LIFO(i))>0 faire
9:              CP(p,i) ← LIFO(i,1)
10:             LIFO(i) ← LIFO(i)\CP(p,i)
11:         Fin si
12:         i ← i+1
13:         Si k<i faire
            /*Recherche du dernier chemin non exploré*/
14:             j ← k
15:             Tant que j>0 faire
16:                 Si Card(LIFO(j))>0 faire
17:                     CP(p+1,1:j-1) ← CP(p,1:j-1)
18:                     CP(p+1,j) ← LIFO(j,1)
19:                     LIFO(j) ← LIFO(j)\CP(p+1,j)
20:                     p ← p+1
21:                     i ← j+1
22:                     j ← 0
23:                 Sinon
24:                     j ← j-1
25:                 Fin si
26:             Fin Tant que
27:         Fin si
28:     Fin Tant que
29: Fin pour chaque
```

Annexe C. Source immédiate

L'algorithme 2 suivant est une application directe des définitions ensemblistes développées dans la section 3.2.2 au résultat de l'algorithme de l'annexe B. Dans cet algorithme les sources qui précèdent de façon immédiate les nœuds contaminés sont identifiées. Son fonctionnement a été expliqué dans la section 3.3.3.

On rappel que les ensembles SEI_k et SCI_k, sont des ensembles vides au début de l'algorithme, et chaque *FIFO(c), c∈V*, est initialisé à 0. L'ensemble C_k contient tous les nœuds de tous les chemins CP précédemment identifiés.

$d_{min}(LM, c)=\min\{ i \in \mathbb{N}^* / CP(p, i) = c, p \in \mathbb{N}^*\} - 1,$

$DC_k = \{ c \in C_k / d_{min}(LM, c) = k\}.$

$NC_k = V \setminus C_k.$

Algorithme 2: détermination des sources immédiates

```
1:  Pour chaque nœud c∈Ck faire
2:     Si Card(Ip(c))>1 faire
3:         SEIk ← SEIk∪(Ip(c)∩(NCk∪DCk))
4:         SCIk ← SCIk∪{ic∈Ip(c)∩Ck\DCk /dmin(LM,ic)≥dmin(LM,c)}
5:         Pour chaque nœud ic∈(SEIk∪SCIk)∩Ip(c) faire
6:              FIFO(ic) ← Max(FIFO(ic),k-dmin(MO,c)+1)
7:          Fin pour chaque
8:     Fin Si
9:  Fin pour chaque
```

Annexe D. Optimisation matérielle

Cette annexe est consacrée à l'étape d'optimisation présentée dans la section 3.3.4. L'algorithme de ces opérations est complexe à détailler.

Retenons donc les étapes principales :

1 – Déterminer les prédécesseurs de toutes les sources immédiates (*Algorithme 3*)

2 – Lister les prédécesseurs qui apparaissent plusieurs fois

3 – Déterminer pour chaque point commun, le ou les remplacements qu'ils permettent (*Algorithme 4*)

4 – Créer pour chaque solution déjà listée une nouvelle solution en effectuant les remplacements permis (*Algorithme 5*).

Pré requis : une première étape consiste pour chaque nœud appartenant à l'ensemble des sources immédiates, à déterminer l'ensemble de leurs prédécesseurs. Cette étape, contrairement à *l'algorithme 1*, est une exploration en profondeur mais dans le sens opposé à celui des arcs et sans limitation quant à la profondeur de l'exploration. Pour chaque nœud s de SEI_k et SCI_k on a un tableau antécédent à trois dimensions $Ant_s(p,q,r)$:

- p est l'indice de profondeur,
- q l'indice des nœuds répertoriés au même niveau de profondeur,
- $r \in \{1, 2\}$. $Ant_s(p, q,1)$ contient le $q^{ème}$ nœud situé au niveau de profondeur p du nœud s. $Ant_s(p, q, 2)$ contient le composant situé en sortie du nœud $Ant_s(p, q,1)$ faisant partie du chemin acyclique appartenant à AP_p, permettant de gagner le nœud s à partir du nœud $Ant_s(p, q,1)$. D'une certaine manière, cela s'apparente à une liste chaînée.

Notation particulière :

pr : la variable *pr* désigne le nœud prédécesseur suivant à explorer ;

Liste_redondant : stocke tous les nœuds parcourus à partir d'une source pour éviter de traiter deux fois le même nœud.

Algorithme 3 : Recherche des antécédents des sources immédiates

```
1: Pour chaque nœud s∈SEI_k∪SCI_k faire
2:     pr ← s
3:     p ← 1
4:     Stop ← 0
5:     Liste_redondant ← pr
6:     Ant_s(p, : , 1:2) ← [Ip(pr) s]
7:     Tant que Stop=0 faire
8:        Stop ← 1
9:        q_max ← SIZE(Ant_s(p,:,1))
10:       q_n ← 1
11:       p ← p+1
12:       Pour chaque entier q_a de 1 à q_max
13:           pr ← Ant_s(p-1,q_a,1)
14:           Si chercher(pr dans Liste_redondant)=0 faire
            /*La recherche d'antécédent ne se fait que pour les nœuds
            non explorés*/
15:               Liste_redondant ← Liste_redondant∪{pr}
16:               Si Ip(pr)≠∅ faire
17:                  Stop ← 0
18:                  q ← size(Ip(v))
19:                  Ant_s(p,q_n:q_n+q-1,1) ← Ip(pr)
20:                  Ant_s(p,q_n:q_n+q-1,2) ← pr
21:                  q_n← q_n+q
22:               Fin si
23:           Fin si
24:       Fin Pour chaque
25:    Fin Tant que
26: Fin Pour chaque
```

Il faut ensuite établir la liste L des nœuds qui apparaissent plusieurs fois dans les ensembles *Ant*. Chaque apparition d'un nœud $n \in L$ dans un ensemble *Ant* est alors associé un triplé (s, p_{init}, q_{init}) correspondant à l'emplacement $Ant_s(p_{init}, q_{init}, 1)=n$. n apparaissant plusieurs fois dans les ensembles *Ant*, il est candidat pour effectuer les regroupements tels qu'expliqué dans la section 3.3.4. La liste des nœuds devant remplacer le nœud s est alors déterminée par *l'algorithme 4*.

Notation particulière :

Rep(i, j) est un tableau de deux colonnes, avec *Rep(i, 1)* qui contient le $i^{ème}$ nœud remplaçant s, et *Rep(i, 2)* la FIFO associée.

Algorithme 4 : Recherche des nœuds pour le remplacement d'un nœud s par un nœud n

```
1:  p ← p_init
2:  q ← q_init
3:  pr ← n
4:  Rep(1,:) ← [pr, FIFO(s)+p-1]
5:  next_pr ← ANT_s(p,q,2)
6:  Tant que p>1 faire
7:     p ← p-1
8:     NRep ← [Ip(Next_pr)\{pr}, FIFO(s)+p-1]
9:     Rep ← Rep∪NRep
10:    pr ← next_pr
11:    q ← chercher(pr dans ANT_s(p,:,1))
12:    next_pr ← ANT_s(p,q,2)
13: Fin Tant que
```

Une fois que la liste *Rep* est constituée pour un nœud *n* et *s* donné, les remplacements permettant de passer d'une solution *S* à une solution *S'* se font suivant *l'algorithme 5*. Dans cette algorithme, les notations $FIFO_S(n)$ et $FIFO_{S'}(n)$ sont là pour faire la distinction entre la FIFO attribuée à un nœud *n* en tant que source de l'ensemble solution *S* ou en tant que source de l'ensemble *S'*.

Il faut ensuite répéter le processus pour chaque nœud de *S* (ou maintenant de *S'*) dont le nœud *n* est le prédécesseur. De plus comme tous les nœuds remplaçant le nœud *s* sont inclus dans le tableau Ant_s leurs prédécesseurs sont déjà connus, et figure dans le tableau Ant_s. Il n'est donc pas nécessaire de recalculer un nouveau tableau pour chacun de ces nœuds, les informations contenues dans le tableau Ant_s suffisent.

Algorithme 5 : Remplacement d'un nœud s par une liste de nœud Rep

```
1:  S'← S\{s}
2:  Pour chaque i de 1 à SIZE(Rep(:,1)) faire
3:       Si Rep(i,1)∈S faire
4:          FIFO_S'(Rep(i,1))=MAX(FIFO_S(Rep(i,1)),(Rep(i,2))
5:       Sinon
6:          S' ← S∪ Rep(i,1)
7:          FIFO_S'(Rep(i,1)) ← (Rep(i,2))
8:       Fin Si
9:  Fin Pour chaque
```

Annexe E. Cycle de décontamination

Cette annexe présente l'algorithme détaillé, établissant les différentes étapes de la décontamination. Rappelons que dans la section 3.2.3 et 3.3.5 on a précisé les cycles de décontamination durant lesquels les FIFO sont actives, lesquels sont liés à la taille des FIFO. Les cycles d'horloge sont comptés à partir du moment où les données corrigées sont émises. La décontamination démarre au cycle $k - max(FIFO(s), s \in S) + 1$, et finit au $k^{ème}$ cycle. Rappelons également que, lorsqu'un numéro de cycle d'horloge est inférieur à 1, cela signifie qu'il s'agit d'une étape devant précéder l'émission de la donnée corrigée par le code correcteur. Comme précisé dans le chapitre 2, tous les composants de stockage sont d'abord bloqués. A *clk=1*, les données corrigées sont fournies par le circuit de correction. A chaque cycle, l'état des bascules débloquées est mis à jour.

Le principe pour déterminer les cycles de décontamination repose sur une idée simple déjà développée dans le chapitre 2 : une fonction logique, durant un cycle de décontamination, fournit un résultat valide si toutes ses entrées sont valides. Le résultat valide est alors stocké dans un registre, lequel doit être débloqué durant ce cycle pour enregistrer le résultat. Au cycle suivant, il fournira une donnée valide. Du point de vue de la représentation sous forme de graphe, un nœud *v* est considéré comme valide durant un cycle *clk,* si au cycle *clk–1* tous les nœuds *Ip(v)* sont valides. Une exception est faite pour les nœuds sources. Un nœud source fournit une donnée valide si celui-ci est décontaminé au cycle précédent (pour les sources contaminables), ou si la FIFO associée est active. Dans le cas d'une source contaminable, rappelons que la FIFO associée n'est plus active dès que la source est décontaminée. De cette façon on peut déterminer cycle par cycle les FIFO qui sont actives, et les registres qui doivent être débloqués (*Algorithme 6*).

Notations particulières :

c.lock : la fonction indiquant si le composant *c* est verrouillé ou non.

c.valid : la fonction indiquant si la donnée provenant de *c* est valide ou non.

c.fifo_lock : la fonction indiquant si la FIFO associée à *c*, pour les nœuds qui ont une FIFO associée est active ou non.

Algorithme 6: Cycle de décontamination

```
    /*Initialisation*/
1:  Pour chaque nœud c ∈ V faire
2:     c.lock ← 1
3:     c.valid ← 0
4:     c.fifo_lock ← 1
5:  Fin Pour chaque
    /*Etapes de la décontamination*/
6:  Pour chaque cycle clk de k-max(FIFO(s), s∈S)+1 à k faire
7:     Pour chaque nœud c ∈ SES_k ∪ SCS_k faire
8:        Si FIFO(c) = k+1-clk faire
9:           c.valid ← 1
10:          c.fifo_lock ← 0
11:       Fin Si
12:    Fin Pour chaque
       /*Quels sont les registres devenus valides ?*/
13:    Pour chaque nœud c ∈ V faire
14:       Si c.lock = 0 faire
15:          c.valid ← 1
16:          c.fifo_lock ← 1
17:       Fin Si
18:    Fin Pour chaque
       /*Débloque les registres qui peuvent l'être*/
19:    Pour chaque nœud c ∈ V faire
20:       Si c.lock = 1 faire
21:          c.lock ← 0
22:          Pour chaque nœud c'∈Ip(c) faire
23;             Si c'.valid = 0 faire
24:                c.lock ← 1
25:             Fin Si
26:          Fin Pour chaque
27:       Fin Si
28:    Fin Pour chaque
29: Fin Pour chaque
    /*Redémarrage du circuit*/
30: Pour chaque nœud c ∈ V faire
31:     c.lock ← 0
32:     c.valid ← 1
33:     c.fifo_lock ← 0
34: Fin Pour chaque
```

Annexe F. Marquage des nœuds

Cette annexe concerne l'algorithme permettant de marquer les nœuds d'un graphe orienté pour :

1 – identifier les nœuds sources

2 – décomposer le circuit représenté sous forme de graphe en pipeline régulier.

Pour déterminer les pipelines réguliers nous employons une variable de marquage triple appliquée à chaque nœud n du circuit $f(n)=(S_o(n), d_s(n), S_r(n))$:

- Lorsque $S_o(n)=1$ (source) le nœud est identifié comme étant un nœud source, $S_o(n)=0$ sinon.
- La valeur $d_s(n)$ est un entier qui indique dans le cas où $S_o(n)=0$ à quelle distance se trouve le nœud n du premier nœud source. On ne peut pas utiliser $d_s(n)=0$ pour identifier un nœud source, car c'est la valeur par défaut attribuée à chaque nœud au début de l'algorithme
- La variable $S_r(n)$ est l'ensemble de nœuds sources desquels le nœud *n* dépend.

Compte tenu qu'il s'agit d'identifier des pipelines réguliers, tous les nœuds doivent vérifier à la fin de l'algorithme de marquage les propriétés 1 et 2, qui se traduisent par :

$$\forall n \in V, \exists! d \in N, d = d_s(n) / \forall s \in S_r(n), d_{min}(s,n) = d$$

$$\forall n \in V, S_O(n) = 0, \exists! d \in N, d_S(n) = d / \forall n' \in Ip(n), d_S(n') = d - 1$$

Notations particulières :

S : représente, comme dans le chapitre 3 l'ensemble des nœuds sources. Cet ensemble contient au début de l'algorithme tous les nœuds du graphe sans prédécesseur.

LIPO(i) : La pile contenant tous les composants connectés en sortie du i–1ème composant du chemin exploré actuellement.

del : contient l'ensemble des nœuds successeurs à un nœud qui vient d'être identifié comme un nœud source et dont les variables doivent être réinitialisées.

MARQUE : cette fonction récursive est chargée d'attribuer les variables de marquage à chaque nœud c du graphe

ROLL : est la fonction qui permet de revenir au dernier chemin non exploré relatif au nœud source traité actuellement

EFFACE : cette fonction récursive réinitialise tous les nœuds successeurs d'un nœud venant d'être identifié comme un nœud source.

Algorithme 7a : Marquage des nœuds, la fonction principale et la fonction MARQUE

```
/*Fonction principale*/
1:    Pour chaque nœud s ∈ S faire
2:          c ← s
3:          d ← 0
4:          LIFO(d+1) ← Op(c)
5:          Si Card(LIFO(d+1))>0 faire
6:                c ← LIFO(d+1,1)
7:                LIFO(d+1) ← LIFO(d+1)\{c}
8:                MARQUE(c)
9:          Fin Si
10:   Fin Pour chaque nœud
/*Fonction de marquage*/
Fonction MARQUE(c : nœud)
1:    d ← d+1
      /*c est un nœud source traité par la fonction principale*/
2:    Si So(c)=1 faire
3:          ROLL(c)
4:    Sinon
            /*Si le pipeline est régulier, le nœud est marqué*/
5:          Si ds(c)=0 OU ds(c)=d faire
6:                ds(c) ← d
7:                Sr(c) ← Sr(c)∪{s}
8:                LIFO(d+1) ← Op(c)
9:                Si Card(LIFO(d+1))>0 faire
10:                     c ← LIFO(d+1,1)
11:                     LIFO(d+1) ← LIFO(d+1)\{c}
12:                     MARQUE(c)
13:               Fin Si
14:         Sinon
                  /*marquage d'un nœud comme un nœud source*/
15:               So(c) ← 1
16:               S ← S∪{c}
17:               e ← c
18:               EFFACE(e)
19:               ROLL(c)
20:         Fin Si
21:   Fin Si
Fin fonction
```

Algorithme 7b: Marquage des nœuds, la fonction EFFACE et la fonction ROLL

```
/*Fonction qui efface le marquage des nœuds suivants un nœud
source nouvellement identifié*/
Fonction EFFACE(e : nœud)
1:    ds(e) ← 0
2:    Sr(e) ← ∅
3:    Pour chaque nœud v∈Ip(e) faire
            /*Dans le cas ou un nœud à effacer est au confluent
            de  plusieurs chemins déjà traité*/
4:          Si ds(v)>0 faire
5:                Sо(e) ← 1
6:          Fin Si
7:    Fin Pour chaque
8:    del ← del ∪ {v∈Op(e) / ds(v)>0}
9:    Si Card(del)>0 faire
10:         e ← del(1)
11:         del ← del\{e}
12:         EFFACE(e)
13:   Fin Si
Fin fonction

/*Fonction revient au dernier chemin non traité*/
Fonction ROLL(c)
1:    Tant que d>0 ET Card(LIFO(d))=0
2:          d ← d-1
3:    Fin Tant que
4:    Si d>0 faire
5:          d ← d-1
6:          c ← LIFO(d+1,1)
7:          LIFO(d) ← LIFO(d+1)\{c}
8:          MARQUE(c)
9:    Fin Si
Fin fonction
```

Bibliographie

[Ale11] Alexandrescu, D.; , "A comprehensive soft error analysis methodology for SoCs/ASICs memory instances," *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International* , vol., no., pp.175-176, 13-15 July 2011.

[ASS09] Avirneni, N.D.P.; Subramanian, V.; Somani, A.K.; , "Low overhead Soft Error Mitigation techniques for high-performance and aggressive systems," *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on* , vol., no., pp.185-194, June 29 2009-July 2 2009.

[Bau05] Baumann, R.C.; , "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on* , vol.5, no.3, pp. 305- 316, Sept. 2005.

[BB97] Baze, M.P.; Buchner, S.P.; , "Attenuation of single event induced pulses in CMOS combinational logic," *Nuclear Science, IEEE Transactions on* , vol.44, no.6, pp.2217-2223, Dec 1997.

[BBB+97] Buchner, S.; Baze, M.; Brown, D.; McMorrow, D.; Melinger, J.; , "Comparison of error rates in combinational and sequential logic," *Nuclear Science, IEEE Transactions on* , vol.44, no.6, pp.2209-2216, Dec 1997.

[BBN+07] Bajura, M.A.; Boulghassoul, Y..; Naseer, R..; DasGupta, S..; Witulski, A.F.; Sondeen, J..; Stansberry, S.D.; Draper, J..; Massengill, L.W.; Damoulakis, J.N.; , "Models and Algorithmic Limits for an ECC-Based Approach to Hardening Sub-100-nm SRAMs," *Nuclear Science, IEEE Transactions on* , vol.54, no.4, pp.935-945, Aug. 2007.

[BD07] Blum, D.R.; Delgado-Frias, J.G.; , "Hardened by Design Techniques for Implementing Multiple-Bit Upset Tolerant Static Memories," *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on* , vol., no., pp.2786-2789, 27-30 May 2007.

[BDS03] Barth, J.L.; Dyer, C.S.; Stassinopoulos, E.G.; , "Space, atmospheric, and terrestrial radiation environments," *Nuclear Science, IEEE Transactions on* , vol.50, no.3, pp. 466- 482, June 2003.

[BEA+04] Benedetto, J.; Eaton, P.; Avery, K.; Mavis, D.; Gadlage, M.; Turflinger, T.; Dodd, P.E.; Vizkelethyd, G.; , "Heavy ion-induced digital single-event transients in deep submicron Processes," *Nuclear Science, IEEE Transactions on* , vol.51, no.6, pp. 3480- 3485, Dec. 2004.

[BHM+95] Baumann, R.; Hossain, T.; Murata, S.; Kitagawa, H.; , "Boron compounds as a dominant source of alpha particles in semiconductor devices," *Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International*, vol., no., pp.297-302, 4-6 April 1995.

[BKL+08] Blaauw, D.; Kalaiselvan, S.; Lai, K.; Wei-Hsiang Ma; Pant, S.; Tokunaga, C.; Das, S.; Bull, D.; , "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance," *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, vol., no., pp.400-622, 3-7 Feb. 2008.

[BN98] Bradley, P.D.; Normand, E.; , "Single event upsets in implantable cardioverter defibrillators," *Nuclear Science, IEEE Transactions on* , vol.45, no.6, pp.2929-2940, Dec 1998.

[BPN+06] Belhaddad, H.; Perez, R.; Nicolaidis, M.; Gaillard, R.; Derbey, M.; Benistant, F.; , "Circuit Simulations of SEU and SET Disruptions by Means of an Empirical Model Built Thanks to a Set of 3D Mixed-Mode Device Simulation Responses", *Proceedings of RADECS*, 2006.

[BSH75] Binder, D.; Smith, E. C.; Holman, A. B.; , "Satellite Anomalies from Galactic Cosmic Rays," *Nuclear Science, IEEE Transactions on* , vol.22, no.6, pp.2675-2680, Dec. 1975.

[BTN+09] Bowman, K.A.; Tschanz, J.W.; Nam Sung Kim; Lee, J.C.; Wilkerson, C.B.; Lu, S.-L.L.; Karnik, T.; De, V.K.; , "Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance," *Solid-State Circuits, IEEE Journal of* , vol.44, no.1, pp.49-63, Jan. 2009.

[CH84] Chen, C. L.; Hsiao, M. Y.; , "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," *IBM Journal of Research and Development* , vol.28, no.2, pp.124-134, March 1984.

[Cis03] Cisco "Cisco 12000 Single Event Upset Failures Overview and Work Around Summary", April15, 2003. http://www.cisco.com/en/US/ts/fn/200/fn25994.html.

[CM08] Choudhury, M.R.; Mohanram, K.; , "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," *Design, Automation and Test in Europe, 2008. DATE '08*, vol., no., pp.903-908, 10-14 March 2008.

[CMF+98] Campbell, A.B.; Musseau, O.; Ferlet-Cavrois, V.; Stapor, W.J.; McDonald, P.T.; , "Analysis of single event effects at grazing angle [CMOS SRAMs]," *Nuclear Science, IEEE Transactions on* , vol.45, no.3, pp.1603-1611, Jun 1998.

[CMN+08] Chang, L.; Montoye, R.K.; Nakamura, Y.; Batson, K.A.; Eickemeyer, R.J.; Dennard, R.H.; Haensch, W.; Jamsek, D.; , "An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches," *Solid-State Circuits, IEEE Journal of* , vol.43, no.4, pp.956-963, April 2008.

[CNV96] Calin, T.; Nicolaidis, M.; Velazco, R.; , "Upset hardened memory design for submicron CMOS technology," *Nuclear Science, IEEE Transactions on* , vol.43, no.6, pp.2874-2878, Dec 1996.

[CRS+05] Chang, J.; Rusu, S.; Shoemaker, J.; Tam, S.; Ming Huang; Haque, M.; Siufu Chiu; Kevin Truong; Karim, M.; Leong, G.; Desai, K.; Goe, R.; Kulkarni, S.; , "A 130-nm triple-Vt 9-MB third-level on-die cache for the 1.7-GHz Itanium® 2 processor," *Solid-State Circuits, IEEE Journal of* , vol.40, no.1, pp. 195- 203, Jan. 2005.

[CVN95] Calin, T.; Vargas, F.L.; Nicolaidis, M.; , "Upset-tolerant CMOS SRAM using current monitoring: prototype and test experiments," *Test Conference, 1995. Proceedings., International*, vol., no., pp.45-53, 21-25 Oct 1995.

[DNB+98] Duarte, R. O.; Nicolaidis, M.; Bederr, H.; Zorian, Y.; , "Efficient fault-secure shifter design, " *J. Electron. Test., Theory Appl.*, vol. 12, no. 1–2, pp. 29–39, Feb. 1998.

[DNR02] Dupont, E.; Nicolaidis, M.; Rohr, P.; , "Embedded robustness IPs for transient-error-free ICs," *Design & Test of Computers, IEEE* , vol.19, no.3, pp.54-68, May/Jun 2002.

[DT07] Dutta, A.; Touba, N.A.; , "Multiple Bit Upset Tolerant Memory Using a Selective Cycle Avoidance Based SEC-DED-DAEC Code," *VLSI Test Symposium, 2007. 25th IEEE* , vol., no., pp.349-354, 6-10 May 2007.

[DW11] Dixit, A.; Wood, A.; , "The impact of new technology on soft error rates," *Reliability Physics Symposium (IRPS), 2011 IEEE International* , vol., no., pp.5B.4.1-5B.4.7, 10-14 April 2011.

[END+03] Ernst, D.; Nam Sung Kim; Das, S.; Pant, S.; Rao, R.; Toan Pham; Ziesler, C.; Blaauw, D.; Austin, T.; Flautner, K.; Mudge, T.; , "Razor: a low-power pipeline based on circuit-level timing speculation," *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* , vol., no., pp. 7- 18, 3-5 Dec. 2003.

[FNM09] Fazeli, M.; Namazi, A.; Miremadi, S.G.; , "An energy efficient circuit level technique to protect register file from MBUs and SETs in embedded processors," *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on* , vol., no., pp.195-204, June 29 2009-July 2 2009.

[For00] Forbes Magazine: Daniel Lyons, “Sun Screen”, Nov. 2000. http://members.forbes.com/global/2000/1113/0323026a.html.

[GBT05] Ghosh, S.; Basu, S.; Touba, N.A.; , "Selecting Error Correcting Codes to Minimize Power in Memory Checker Circuits", *J. Low Power Electronics* 1, pp.63-72 (2005).

[GGR+04] Gordon, M.S.; Goldhagen, P.; Rodbell, K.P.; Zabel, T.H.; Tang, H.H.K.; Clem, J.M.; Bailey, P.; , "Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground," *Nuclear Science, IEEE Transactions on* , vol.51, no.6, pp. 3427- 3434, Dec. 2004.

[GJK+06] Garg, R.; Jayakumar, N.; Khatri, S.P.; Choi, G.; , "A design approach for radiation-hard digital electronics," *Design Automation Conference, 2006 43rd ACM/IEEE*, vol., no., pp.773-778, 0-0 0.

[GNP05] Gill, B.; Nicolaidis, M.; Papachristou, C.; , "Radiation induced single-word multiple-bit upsets correction in SRAM," *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, vol., no., pp. 266- 271, 6-8 July 2005.

[GNW+05] Gill, B.; Nicolaidis, M.; Wolff, F.; Papachristou, C.; Garverick, S.; , "An efficient BICS design for SEUs detection and correction in semiconductor memories," *Design, Automation and Test in Europe, 2005. Proceedings*, vol., no., pp. 592- 597 Vol. 1, 7-11 March 2005.

[Gor94] O'Gorman, T. J.; , "The effect of cosmic rays on the soft error rate of a DRAM at ground level." *IEEE Trans. Elec. Dev.*, 41(4): 553-557, April 1994.

[GR68] Garcia, O. N. and Rao, T. R. N. "On the Methods of Checking Logical Operations," Proc. *2nd Annual Princeton Conference on Information Sciences and Systems*, 1968, pp. 89-95.

[Gra00] Gray, K.; , "Adding error-correcting circuitry to ASIC memory," *Spectrum, IEEE* , vol.37, no.4, pp.55-60, Apr 2000.

[HLS08] Hill, E.L.; Lipasti, M.H.; Saluja, K.K.; , "An accurate flip-flop selection technique for reducing logic SER," *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* , vol., no., pp.128-136, 24-27 June 2008.

[Hsi70] Hsiao, M. Y.; , "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development* , vol.14, no.4, pp.395-401, July 1970.

[Ibe08] Ibe, E.;, "Terrestrial Neutron-Induced Soft Errors In Advanced Memory Devices," *World Scientific Publishing Co.* Pte. Ltd., chapter 1, Mars 2008, http://www.worldscibooks.com/etextbook/6661/6661_chap01.pdf.

[ICS+06] Ibe, E.; Chung, S.S.; Wen, S. J.; Yamaguchi; H.; Yahagi, Y.; Kameyama, H.; Yamamoto, S.; Akioka, T.; , "Spreading Diversity in Multi-cell Neutron-Induced Upsets with Device Scaling," *Custom Integrated Circuits Conference, 2006. CICC '06. IEEE* , vol., no., pp.437-444, 10-13 Sept. 2006.

[Int05] Intel. Excerpts from A Conversation with Gordon Moore: Moore's Law. 2005.

[ITRS09] ITRS: International Technology Roadmap for Semiconductors. *In SystemDrivers*, 2009. http://www.itrs.net.

[ITY+09] Ibe, E.; Taniguchi, H.; Yahagi, Y.; Shimbo, K.; Toba, T.; , "Scaling Effects on Neutron-Induced Soft Error in SRAMs Down to 22nm Process," *Third Workshop on Dependable and Secure Nanocomputing*, June 29, 2009, Estoril, Lisbon, Portugal, No.2.1 (2009).

[ITY+10] Ibe, E.; Taniguchi, H.; Yahagi, Y.; Shimbo, K.-i.; Toba, T.; , "Impact of Scaling on Neutron-nduced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *Electron Devices, IEEE Transactions on* , vol.57, no.7, pp.1527-1538, July 2010.

[JED89] JEDEC, "Measurement and Reporting of Alpha Particles and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices: JESD89A," *JEDEC STANDARD, JEDEC Solid State Technology Association*, pp. 1–85, 2006, No. 89.

[KHN+04] Kawakami, Y.; Hane, M.; Nakamura, H.; Yamada, T.; Kumagai, K.; , "Investigation of soft error rate including multi-bit upsets in advanced SRAM using neutron irradiation test and 3D mixed-mode device simulation," *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International* , vol., no., pp. 945- 948, 13-15 Dec. 2004.

[LKN+07] Lisboa, C.A.; Kastensmidt, F.L.; Neto, E.H.; Wirth, G.; Carro, L.; , "Using built-in sensors to cope with long duration transient faults in future technologies," *Test Conference, 2007. ITC 2007. IEEE International*, vol., no., pp.1-10, 21-26 Oct. 2007.

[LT70] Langdon, G. G.; Tang, C. K.; , "Concurrent Error Detection for Group Look-ahead Binary Adders," *IBM Journal of Research and Development* , vol.14, no.5, pp.563-573, Sep. 1970.

[LTR+92] Lo, J.-C.; Thanawastien, S.; Rao, T.R.N.; Nicolaidis, M.; , "An SFS Berger check prediction ALU and its application to self-checking processor designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.11, no.4, pp.525-540, Apr 1992.

[LYL09] Sheng Lin; Yong-Bin Kim; Lombardi, F.; , "A novel design technique for soft error hardening of Nanoscale CMOS memory," *Circuits and Systems, 2009. MWSCAS '09. 52nd IEEE International Midwest Symposium on* , vol., no., pp.679-682, 2-5 Aug. 2009.

[MB05] Mariani, R.; Boschi, G.; , "A System Level Approach for Embedded Memory Robustness," *Solid-State Electronics* 49, 2005.

[MJ05] Muck, M.; Javaudin, J.-P.; , "Advanced OFDM Modulators considered in the IST-WINNER Framework for Future Wireless Systems", *14th IST Mobile and Wireless Communications* Submit, 2005.

[Met09] Metra, C.; , "Trading Off Dependability and Cost for Nanoscale High Performance Microprocessors: The Clock Distribution Problem" 2009 *Workshop on Dependable and Secure Nanocomputing*, June 29, 2009, Lisbon Portugal.

[MHZ+03] Maiz, J.; Hareland, S.; Zhang, K.; Armstrong, P.; , "Characterization of multi-bit soft error events in advanced SRAMs," *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, vol., no., pp. 21.4.1- 21.4.4, 8-10 Dec. 2003.

[MOR+96] Musseau, O.; Gardic, F.; Roche, P.; Corbiere, T.; Reed, R.A.; Buchner, S.; McDonald, P.; Melinger, J.; Tran, L.; Campbell, A.B.; , "Analysis of multiple bit upsets (MBU) in CMOS SRAM," *Nuclear Science, IEEE Transactions on* , vol.43, no.6, pp.2879-2888, Dec 1996.

[MSN+00] Makihara, A.; Shindou, H.; Nemoto, N.; Kuboyama, S.; Matsuda, S.; Oshima, T.; Hirao, T.; Itoh, H.; Buchner, S.; Campbell, A.B.; , "Analysis of single-ion multiple-bit upset in high-density DRAMs," *Nuclear Science, IEEE Transactions on* , vol.47, no.6, pp.2400-2404, Dec 2000.

[MSS+07] Meaney, P.J.; Swaney, S.B.; Sanda, P.N.; Spainhower, L.; , "IBM z990 soft error detection and recovery," *Device and Materials Reliability, IEEE Transactions on* , vol.5, no.3, pp. 419- 427, Sept. 2005.

[MW78] May, T. C.; Woods, M. H.; , "A New Physical Mechanism for Soft Errors in Dynamic Memories," *Reliability Physics Symposium, 1978. 16th Annual*, vol., no., pp.33-40, April 1978.

[NBS+07] Narasimham, B.; Bhuva, B.L.; Schrimpf, R.D.; Massengill, L.W.; Gadlage, M.J.; Amusan, O.A.; Holman, W.T.; Witulski, A.F.; Robinson, W.H.; Black, J.D.; Benedetto, J.M.; Eaton, P.H.; , "Characterization of Digital Single Event Transient Pulse-Widths in 130-nm and 90-nm CMOS Technologies," *Nuclear Science, IEEE Transactions on* , vol.54, no.6, pp.2506-2511, Dec. 2007.

[ND99] Nicolaidis, M.; Duarte, R.O.; , "Fault-secure parity prediction Booth multipliers," *Design & Test of Computers, IEEE* , vol.16, no.3, pp.90-101, 1999.

[NDM+97] M. Nicolaidis, R. O. Duarte, S. Manich, and J. Figueras, "Achieving fault secureness in parity prediction arithmetic operators," *IEEE Des. Test. Comput.*, vol. 14, no. 3, pp. 60–71, Apr.–Jun. 1997.

[Nic99] Nicolaidis, M.; , "Time redundancy based soft-error tolerance to rescue nanometer technologies," *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, vol., no., pp.86-94, 1999.

[Nic03] Nicolaidis, M.; , "Carry checking/parity prediction adders and ALUs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.11, no.1, pp.121-128, Feb. 2003.

[Nic05] Nicolaidis, M.; , "Design for soft error mitigation," *Device and Materials Reliability, IEEE Transactions on* , vol.5, no.3, pp. 405- 418, Sept. 2005.

[Nic07] Nicolaidis, M.; , "GRAAL: a new fault tolerant design paradigm for mitigating the flaws of deep nanometric technologies," *Test Conference, 2007. ITC 2007. IEEE International* , vol., no., pp.1-10, 21-26 Oct. 2007.

[Nor96] Normand, E.; , "Single event upset at ground level," *Nuclear Science, IEEE Transactions on* , vol.43, no.6, pp.2742-2750, Dec 1996.

[OBF+93] Olsen, J.; Becher, P.E.; Fynbo, P.B.; Raaby, P.; Schultz, J.; , "Neutron-induced single event upsets in static RAMS observed a 10 km flight attitude," *Nuclear Science, IEEE Transactions on* , vol.40, no.2, pp.74-77, Apr 1993.

[OGS+03] Ocheretnij, V.; Gossel, M.; Sogomonyan, E.S.; Marienfeld, D.; , "A modulo p checked self-checking carry select adder," *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, vol., no., pp. 25- 29, 7-9 July 2003.

[OMS+04] Ocheretnij, V.; Marienfeld, D.; Sogomonyan, E.S.; Gossel, M.; , "Self-checking code-disjoint carry-select adder with low area overhead by use of add1-circuits," *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, vol., no., pp. 31- 36, 12-14 July 2004.

[Pet58] Peterson, W. W.; , "On Checking an Adder," *IBM Journal of Research and Development* , vol.2, no.2, pp.166-168, Apr. 1958.

[Pra96] Pradhan, D. K.; *Fault-Tolerant Computing System Design*. Upper Saddle River, NJ: Prentice-Hall, 1996.

[PW72] Peterson W. W.; Weldon, E. J.; *Error-Correcting Codes*, 2nd ed. Cambridge, MA: MIT Press, 1972.

[RBB+11] Riedlinger, R.J.; Bhatia, R.; Biro, L.; Bowhill, B.; Fetzer, E.; Gronowski, P.; Grutkowski, T.; , "A 32nm 3.1 billion transistor 12-wide-issue Itanium® processor for mission-critical servers," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International* , vol., no., pp.84-86, 20-24 Feb. 2011.

[RPS+05] Radaelli, D.; Puchner, H.; Skip Wong; Daniel, S.; , "Investigation of multi-bit upsets in a 150 nm technology SRAM device," *Nuclear Science, IEEE Transactions on* , vol.52, no.6, pp. 2433-2437, Dec. 2005.

[RRV+02] Rebaudengo, M.; Reorda, M. S.; Violante, M.; Nicolescu, B.; Velazco, R.; "Coping with SEUs/SETs in microprocessors by means of low-cost solutions: A comparative study," *IEEE Trans. Nucl. Sci.*, vol. 49, no. 3, pp. 1491–1495, Jun. 2002.

[Sah09] Sahnine, C.; "Architecture de circuit intégré reconfigurable, très haut débit et basse consommation pour le traitement numérique de l'OFDM avancé", *thèse de doctorat,* Grenoble INP, France, 2009.

[SGB10] Sivamangai, N.M.; Gunavathi, K. ; Balakrishnan, P.; , " A BICS Design to Detect Soft Error in CMOS SRAM," *International Journal on Computer Science and Engineering* Vol. 02, No. 03, 2010, pp. 734-740.

[SHB68] Sellers, F. F.; Hsiao, M.-Y.; Bearnson, L. W.; *Error Detecting Logic for Digital Computers*. New York: McGraw-Hill, 1968.

[SKF01] Shyh-Kwei Chen; Fuchs, W.K.; , "Compiler-assisted multiple instruction word retry for VLIW architectures," *Parallel and Distributed Systems, IEEE Transactions on* , vol.12, no.12, pp.1293-1304, Dec 2001.

[SP88] Smith, J.E.; Pleszkun, A.R.; , "Implementing precise interrupts in pipelined processors," *Computers, IEEE Transactions on* , vol.37, no.5, pp.562-573, May 1988.

[SS92] Siewiorek, D. P.; Swwarz, R. S.; *Reliable Computer Design and Evaluation*. Newton, MA: Digital, 1992.

[SSK+06] Seifert, N.; Slankard, P.; Kirsch, M.; Narasimham, B.; Zia, V.; Brookreson, C.; Vo, A.; Mitra, S.; Gill, B.; Maiz, J.; , "Radiation-Induced Soft Error Rates of Advanced CMOS Bulk Devices," *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, vol., no., pp.217-225, 26-30 March 2006.

[SSW09] Sanghyeon Baeg; ShiJie Wen; Wong, R.; , "SRAM Interleaving Distance Selection With a Soft Error Failure Model," *Nuclear Science, IEEE Transactions on* , vol.56, no.4, pp.2111-2118, Aug. 2009.

[SVC09] Sinangil, M.E.; Verma, N.; Chandrakasan, A.P.; , "A Reconfigurable 8T Ultra-Dynamic Voltage Scalable (U-DVS) SRAM in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of* , vol.44, no.11, pp.3163-3173, Nov. 2009.

[SYL11] Sheng Lin; Yong-Bin Kim; Lombardi, F.; , "Modeling and design of a nanoscale memory cell for hardening to a single event with multiple node upset," *Computer Design (ICCD), 2011 IEEE 29th International Conference on* , vol., no., pp.320-325, 9-12 Oct. 2011.

[TEI+04] Tosaka, Y.; Ehara, H.; Igeta, M.; Uemura, T.; Oka, H.; Matsuoka, N.; Hatanaka, K.; , "Comprehensive study of soft errors in advanced CMOS circuits with 90/130 nm technology," *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, vol., no., pp. 941- 944, 13-15 Dec. 2004.

[TM94] Touba, N.A.; McCluskey, E.J.; , "Logic Synthesis Techniques For Reduced Area Implementation Of Multilevel Circuits With Concurrent Error Detection," *Computer-Aided Design, 1994., IEEE/ACM International Conference on* , vol., no., pp.651-654, 6-10 Nov 1994.

[TNU+09] Tanaka, K.; Nakamura, H.; Uemura, T.; Takeuchi, K.; Fukuda, T.; Kumashiro, S.; , "Study on Influence of Device Structure Dimensions and Profiles on Charge Collection Current Causing SET Pulse Leading to Soft Errors in Logic Circuits," *Simulation of Semiconductor Processes and Devices, 2009. SISPAD '09. International Conference on*, vol., no., pp.1-4, 9-11 Sept. 2009.

[TT89] Tremblay, M.; Tamir, Y.; , "Fault-Tolerance for High-Performance Multi-Module VLSI Systems Using Micro Rollback," Advanced Research in *VLSI: Proceedings of the Decennial Caltech Conference on VLSI* Pasadena, California, March 1989, The MIT Press, pp. 297-316.

[TTR88] Tamir, Y.; Tremblay, M.; Rennels, D. A.; , "The implementation and application of micro rollback in fault-tolerant VLSI systems," in *Proc.18th Fault-Tolerant Comput. Symp.,* Tokyo, Japan, June 1988, pp. 234-239.

[VJA+08] Vemu, R.; Jas, A.; Abraham, J.A.; Patil, S.; Galivanche, R.; , "A low-cost concurrent error detection technique for processor control logic," *Design, Automation and Test in Europe, 2008. DATE '08* , vol., no., pp.897-902, 10-14 March 2008.

[WJB06] Wood, A.; Jardine, R.; Bartlett, W.; , "Data integrity in HP nonstop servers," in *Proc. SELSE II*, Urbana-Champain, IL, Apr. 11–12, 2006.

[WM62] Wallmark, J.T.; Marcus, S.M.; , "Minimum Size and Maximum Packing Density of Nonredundant Semiconductor Devices," *Proceedings of the IRE* , vol.50, no.3, pp.286-298, March 1962.

[YK11] Yu Liu; Kaijie Wu; , "Runtime adaptable concurrent error detection for linear digital systems," *Computer Design (ICCD), 2011 IEEE 29th International Conference on* , vol., no., pp.261-266, 9-12 Oct. 2011.

[ZL79] Ziegler, J. F.; Lanford, W. A.; , "Effect of Cosmic Rays on Computer Memories", *Science*, 206, 776, Nov 1979.

[ZMM+06] Zhang, M.; Mitra, S.; Mak, T. M.; Seifert, N.; Wang, N. J.; Shi, Q.; Kim, K. S.; Shanbhag, N. R.; Patel, S. J.; , "Sequential Element Design With Built-In Soft Error Resilience," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.14, no.12, pp.1368-1378, Dec. 2006.

Publications

Articles publiés lors de conférences internationnales :

- Nicolaidis, M.; Bonnoit, T.; Zergainoh, N.-E.; , "Eliminating speed penalty in ECC protected memories," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* , vol., no., pp.1-6, 14-18 March 2011.
- Bonnoit, T.; Nicolaidis, M.; Zergainoh, N.; , "Towards a tool for implementing delay-free ECC in embedded memories," *Computer Design (ICCD), 2011 IEEE 29th International Conference on* , vol., no., pp.441-442, 9-12 Oct. 2011.

Article de Journal en attente d'acceptation :

- Bonnoit, T.; Nicolaidis, M.; Zergainoh, N.; , "Using Error Correcting Codes without Speed Penalty in Embedded Memories: Algorithm, Implementation and Case Study", Journal of Electronic Testing: Theory and Applications, Special Issue on Defect and Fault Tolerance.

Architectures pour des circuits fiables de hautes performances

Résumé : Les technologies nanométriques ont réduit la fiabilité des circuits électroniques, notamment en les rendant plus sensible aux phénomènes extérieurs. Cela peut provoquer une modification des composants de stockage, ou la perturbation de fonctions logiques. Ce problème est plus préoccupant pour les mémoires, plus sensibles aux perturbations extérieures. Les codes correcteurs d'erreurs constituent l'une des solutions les plus utilisées, mais les contraintes de fiabilité conduisent à utiliser des codes plus complexes, et qui ont une influence négative sur la bande passante du système. Nous proposons une méthode qui supprime la perte de temps due à ces codes lors de l'écriture des données en mémoire, et la limite aux seuls cas où une erreur est détectée lors de la lecture. Pour cela on procède à la décontamination du circuit après qu'une donnée erronée ait été propagée dans le circuit, ce qui nécessite de restaurer certains des états précédents de quelques composants de stockage par l'ajout de FIFO. Un algorithme identifiant leurs lieux d'implémentation a également été créé. Nous avons ensuite évalué l'impact de cette méthode dans le contexte plus large suivant : la restauration d'un état précédent de l'ensemble du circuit en vue de corriger une erreur transistoire susceptible de se produire n'importe où dans le circuit.

Mots clés : mémoires ; codes correcteurs ; décontamination ; sauvegarde de contexte ; erreur transistoires ; restauration d'états.

Architectural solutions for reliable and high performance circuits

Abstract: Nanometric technologies led to a decrease of electronic circuit reliability, especially against external phenomena. Those may change the state of storage components, or interfere with logical components. In fact, this issue is more critical for memories, as they are more sensitive to external radiations. The error correcting codes are one of the most used solutions. However, reliability constraints require codes that are more and more complex. These codes have a negative effect on the system bandwidth. We propose a generic methodology that removes the timing penalty of error correcting codes during memory's write operation. Moreover, it limits the speed penalty for read operation only in the rare case an error is detected. To proceed, the circuit is decontaminated after uncorrected data were propagated inside the circuit. This technique may require restoring some past states of few storage components by adding some FIFO. An algorithm that identifies these components was also created. Then we try to evaluate the impact of such a technique for the following issue: the global state restoration of a circuit to erase all kinds of soft errors, everywhere inside the circuit.

Key words: memories; error correcting codes; decontamination; state preserve mechanism; soft errors; state restoration.

ISBN 978-2-84813-195-5