**LABORATOIRE INTERFACES**

**CAPTEURS ET MICORELECTRONIQUE**

**Ecole Doctorale IAEM – Lorraine**

# THESE DE DOCTORAT

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

Discipline: Electronique

Spécialité: Microélectronique

## Méthodologie de conception d'architectures de processeur sûres de fonctionnement pour les applications mécatroniques

par

**Mehdi JALLOULI**

Soutenance le 04 Juin 2009

Composition du jury

| | | |
|---|---|---|
| Luc **HEBRARD** | Pr., InESS, Université Louis Pasteur de Strasbourg | (Rapporteur) |
| Emmanuel **SIMEU** | MCF HDR, TIMA, Institut Polytechnique de Grenoble | (Rapporteur) |
| Christophe **BOBDA** | Pr., Université de Potsdam, Allemagne | (Examinateur) |
| Abbas **DANDACHE** | Pr., LICM, Université Paul Verlaine de Metz | (Examinateur) |
| Fabrice **MONTEIRO** | Pr., LICM, Université Paul Verlaine de Metz | (Directeur de thèse) |
| Camille **DIOU** | MCF, LICM, Université Paul Verlaine de Metz | (Co-encadrant) |

LICM – 07 rue Marconi, 57070 Metz, France

Tél. +33 (0)3 87 54 73 06 / 11 – Fax : +33 (0)3 87 54 73 07 – www.licm.fr

Fondationcetim
Sous l'égide de la Fondation de France

Ce travail est réalisé dans le cadre du projet CIM'tronic (projet fondation CETIM) : conception intégrée de systèmes mécatroniques sûrs de fonctionnement. Ce projet inclut quatre partenaires, composant le consortium :

- Automatismes et Simulation pour la Sécurité des Systèmes Industriels (A3SI) – ENSAM de Metz (57) – Centre de Recherche en Automatique de Nancy (CRAN) – Nancy (54) – UMR 7039.
- Laboratoire Interfaces Capteurs & Microélectronique (LICM) – Metz (57) – EA 1776.
- Institut d'Électronique, du Solide et des Systèmes (InESS) – Strasbourg (67) – UMR 7163.
- Laboratoire de Physique des Matériaux (LPM) – Nancy (54) – UMR 7556.

# Table des matières

# Introduction

“D’après la dernière étude publiée dans la presse économique, sur 100 projets innovants en mécanique, la majorité sont à l’interface de la mécanique et de l’électronique.” [PEIL02] Fernand Peilloud, Président de THESAME. En effet, tout mouvement mécanique, piloté par une électronique intégrée, est « mécatronique ».

La mécatronique est la combinaison synergique et systémique de la mécanique, de l'électronique, et de l'informatique temps réel. L'intérêt de ce domaine d'ingénierie interdisciplinaire est de concevoir des systèmes automatiques puissants et de permettre le contrôle de systèmes hybrides complexes. L'ingénierie de tels systèmes mécatroniques nécessite la conception simultanée et pluridisciplinaire de trois sous-systèmes :

- Une partie opérative : squelette et muscle du système à dominante mécanique et électromécanique ;
- Une partie commande : intelligence embarquée du système à dominante électronique et informatique temps réel ;
- Une partie interface homme/machine : forme géométrique et dialogue du système à dominante ergonomique et esthétique).

Parmi les disciplines qui y sont impliquées, nous citons l’automatique (contrôle et commande), le génie informatique (logiciel), le génie mécanique, et l’électronique, comme c’est indiqué dans la figure 0.1. La mécatronique est ainsi la symbiose de ces différentes disciplines au service de la conception de produits intégrés. Une approche globale permet aussi de réduire les coûts, d'augmenter la fiabilité et la modularité.

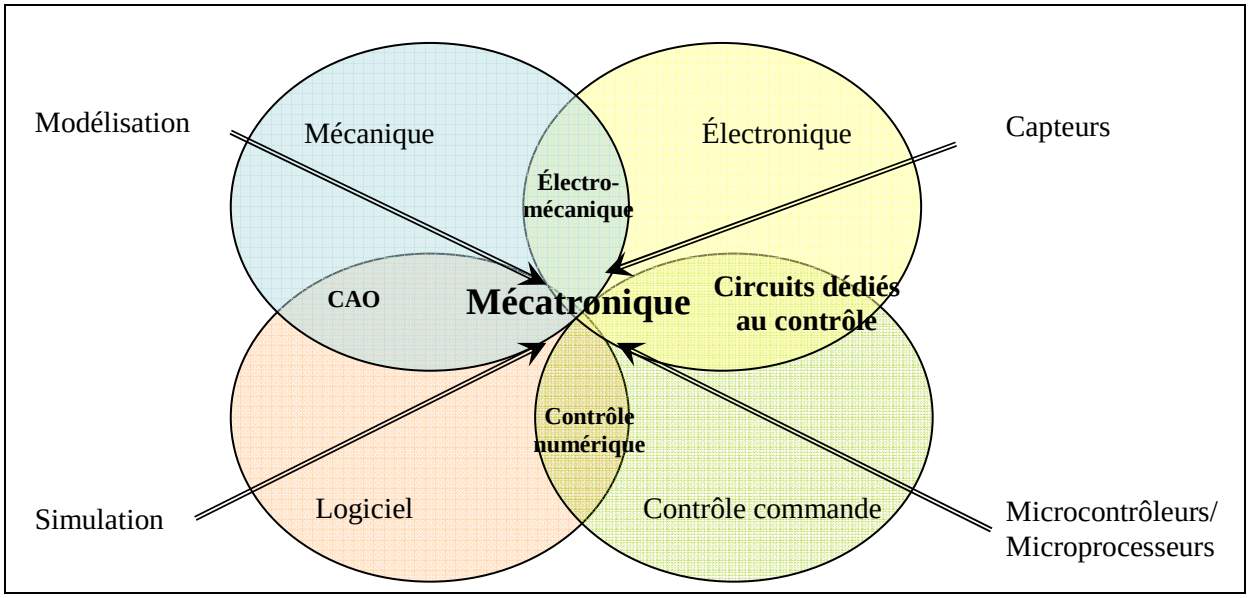


**Figure 0.1 : Interactions entre systèmes et technologies**

Le terme '*mechatronics*' a été introduit par un ingénieur de la compagnie japonaise Yaskawa en 1969. Le terme est apparu officiellement en France dans le Larousse de 2005.

Comme exemples de systèmes mécatroniques, nous pouvons citer :

- Véhicule automobile moderne : système anti-blocage (*ABS : Anti Blocking System*), programme de stabilité électronique (*ESP: Electronic Stability Program*), direction assistée … etc ;
- Avion de chasse ;
- Machine-outil à commande numérique ;
- Disques durs ;
- Machines à laver « intelligentes » ;
- ... etc.

La mécatronique permet l'obtention de fonctionnalités supplémentaires sans surcoût, une diminution du nombre de composants nécessaires à la réalisation d'une fonction, l'amélioration de la maintenabilité et de la disponibilité, la facilité d'intégration, l'amélioration des performances, une diminution des coûts de possession...etc. Malgré l'importance quantitative des travaux scientifiques, de nombreuses problématiques restent à explorer, notamment en terme de sûreté de fonctionnement des systèmes mécatroniques. En effet, l'objectif du projet CIM'tronic (projet fondation CETIM) ne s'arrête pas à la seule conception intégrée de systèmes mécatroniques. Notre but est de lui inclure la sûreté de fonctionnement en contribuant au développement des outils et des mécanismes d'aide à l'évaluation de la sûreté et en définissant les caractéristiques architecturales du cœur

du processeur en vue d'une méthodologie de conception de systèmes mécatroniques sûrs de fonctionnement. C'est pour cette raison, quatre structures reconnues pour leurs compétences complémentaires ont collaboré afin d'atteindre le but global du projet CIM'tronic : conception intégrée de systèmes mécatroniques sûrs de fonctionnement. Une collaboration entre spécialistes de disciplines technologiques différentes est alors établie afin de participer au développement des bases de connaissances nécessaires à la compétitivité future des industries mécaniques. Ces quatre partenaires composant le consortium, sont :

- Automatismes et Simulation pour la Sécurité des Systèmes Industriels (A3SI) – ENSAM de Metz (57) – Centre de Recherche en Automatique de Nancy (CRAN) – Nancy (54) – UMR 7039.
- Laboratoire Interfaces Capteurs & Microélectronique (LICM) – Metz (57) – EA 1776.
- Institut d'Électronique, du Solide et des Systèmes (InESS) – Strasbourg (67) – UMR 7163.
- Laboratoire de Physique des Matériaux (LPM) – Nancy (54) – UMR 7556.

Un intérêt particulier est dédié aux deux points suivants : l'aspect de sûreté de fonctionnement et le contexte mécatronique du projet. En effet, la complexité des capteurs intelligents est liée à la présence de composants matériels et logiciels et à l'interaction entre ces composants. L'existence de signaux mixtes (analogique et numérique) nécessite une approche globale. Les méthodes conventionnelles sont en général dédiées à l'étude de la sûreté du matériel et du logiciel séparément. La sûreté des systèmes mécatroniques nécessite la maîtrise de la disponibilité et de la fiabilité des fonctions de conduite et de sécurité. Une approche transdisciplinaire de conception intégrée de systèmes mécatroniques sûrs de fonctionnement devient ainsi obligatoire.

La figure 0.2 montre les rôles et objectifs de chaque partenaire en tenant compte des collaborations possibles entre les différents partenaires. L'objectif des deux équipes de recherche de l'InESS de Strasbourg et du LPM de Nancy est de modéliser et de maîtriser les processus de vieillissement des composants électroniques pour assurer un niveau de fiabilité accepté. Quant à celle du CRAN de Nancy/A3SI-ENSAM de Metz, c'est la modélisation pour l'analyse fiabiliste et pour une méthodologie de conception d'un capteur intelligent. Enfin, concernant notre laboratoire le LICM de Metz, notre objectif est de proposer et de valider une méthodologie de conception d'une architecture de processeur sûre de fonctionnement ainsi que de concevoir et de développer des outils et des mécanismes de tolérance aux fautes pour

aboutir à une architecture de processeur sûre de fonctionnement. Le développement de tels outils s'inscrit dans un aspect méthodologique. Bien entendu, ceci est fait tout en prenant en compte les interactions matérielles logicielles dès le début de la phase de conception conjointe (*co-design*) [RIEM01], [WIRK03], [WOLF07].

Par ailleurs, un des objectifs importants et fixés par le projet CIM'tronic consistait à intégrer les travaux de divers partenaires en un travail commun. C'est dans ce contexte qu'une partie du travail de cette thèse est dédiée à la validation d'une des approches de sûreté proposée par le LICM en exploitant l'approche du flux informationnel fournie par l'équipe de recherche du CRAN de Nancy/A3SI-ENSAM de Metz.

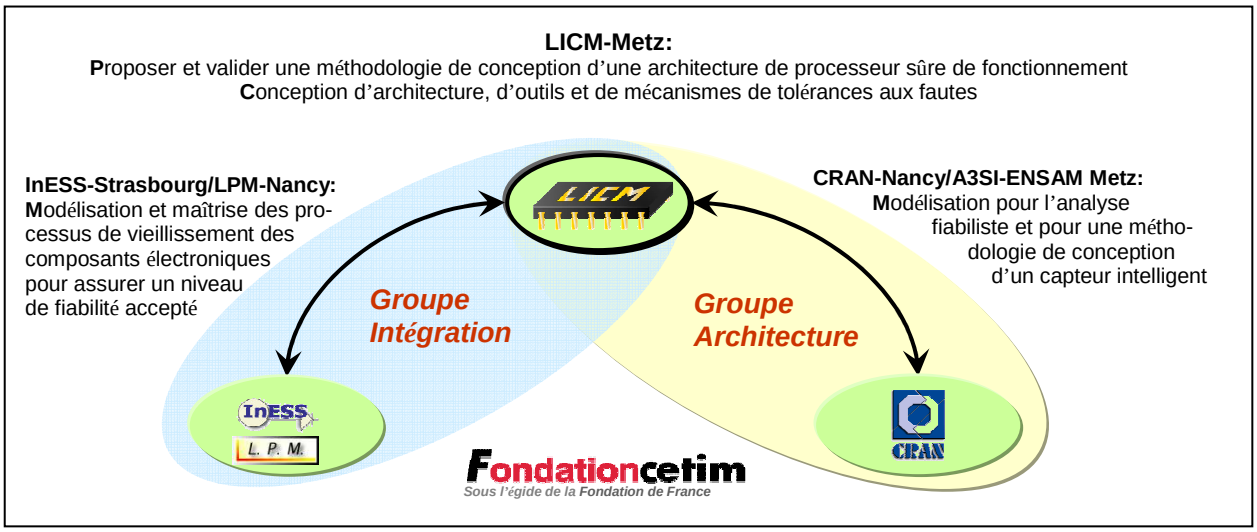


**Figure 0.2 : Interaction entre les membres du consortium**

L'objectif du LICM est de proposer et concevoir des outils d'aide d'évaluation de la sûreté en vue d'une méthodologie de conception d'une architecture de processeur sûre de fonctionnement. Ceci à travers le développement d'outils et de mécanismes de tolérances aux fautes d'une part et la conception et la définition des caractéristiques architecturales du processeur d'autre part. C'est dans ce contexte de sûreté que le premier chapitre portera. Il présente les concepts de base et la terminologie de la sûreté de fonctionnement, en insistant sur les spécificités de sûreté des systèmes embarqués. Ensuite, il met en lumière les défaillances, les erreurs et les fautes pouvant se produire dans de tels systèmes ainsi que des exemples de techniques de recours face à de telles fautes, qu'elles soient des techniques de détection uniquement, ou des techniques de détections et de correction ou également qu'elles soient des techniques de protection logicielles, ou des techniques de protection matérielles. Nous présentons ensuite certains processeurs tolérants aux fautes existants, leurs spécificités et leurs utilités. Nous concluons par leurs propres limites face à notre besoin.

Le second chapitre porte sur l'introduction de la démarche que nous adopterons en vue de dégager une méthodologie de conception d'architecture de processeur sûre de fonctionnement. Il s'agit de montrer les raisons de nos choix en termes de type d'architectures de processeurs et de sa structure interne d'une part, et d'autre part en termes de techniques de protection adoptées, selon une étude d'un compromis logiciel (application) et matériel (processeur). Cette étude est réalisée grâce au développement d'outils logiciels. En premier lieu, un émulateur de processeur, décrivant l'évolution des états internes du processeur, permettra de simuler l'injection d'erreurs. En second lieu, des benchmarks représentatifs de programmes embarqués seront ainsi testés dans l'émulateur. Les tests et validations de cette technique dans l'émulateur feront l'objet du troisième chapitre. Comme notre priorité est la sûreté, nous exploitons ces outils logiciels à travers une simulation d'injection d'erreurs dans l'émulateur et une intégration d'une méthode de protection logicielle dans le *benchmark*. Nous proposons certains scénarii d'apparition d'erreurs afin de comparer les surcoûts temporels de cette technique. Nous concluons en outre sur les différentes capacités de correction selon des spécificités du cahier de charges telles que les contraintes temps-réel ou le taux d'erreur par exemple.

Enfin, le quatrième et dernier chapitre porte sur les résultats obtenus suite à une convergence de nos travaux avec ceux de l'équipe du CRAN de Nancy/A3SI-ENSAM de Metz. Il s'agit de valider la technique de protection décrite dans le deuxième chapitre par le biais d'une approche de type flux informationnel. En effet, des évaluations probabilistes sont destinées à évaluer en termes de probabilités le degré de satisfaction de certains attributs de la sûreté de fonctionnement. L'approche flux informationnel [HAMI05] fournit un ensemble de langages des scénarii de dysfonctionnement, fondés sur un modèle dynamique des automates à états finis, qui sert à l'analyse de cohérence des spécifications et à leur vérification. L'idée consiste à exploiter le chemin de données et le schéma d'exécution de chacune des instructions assembleur afin d'aboutir à leurs modèles de haut niveau et de bas niveau sous forme d'automates à états finis correspondants à la méthode du flux informationnel.

# Références


— [HAMI05] K. Hamidi, O. Malassé and J-F. Aubry: “Coupling of information-flow aggregation method and dynamical model for a more accurate evaluation of reliability”. *European Safety and Reliability Conference* (ESREL05), Poland, June 2005.

— [PEIL02] F. Peilloud, Forum international sur les systèmes à capteurs, la mécatronique, les affaires et l’innovation, Archamps, Haute-Savoie, France, 30 – 31 Janvier 2002.

— [RIEM01] R. A. Riemenschneider and S. Dawson: “Dependability Codesign”, Workshop on *New Visions for Software Design and Productivity*, Nashville, Tennessee, USA, Dec. 2001.

— [WIRK03] N.Wirk: “Hardware/Software co-design then and now”, Information Processing Letters 88 (2003) 83-87.

— [WOLF07] W. Wolf: “Hardware and Software Co-design”, High-Performance Embedded Computing, 2007, pp. 383-432.

# Chapitre 1

# État de l’art

Les systèmes embarqués sont de plus en plus présents dans notre environnement (aéronautique, automobile, énergie, téléphonie mobile, transport ferroviaire...). Interconnectés et communicants, conçus pour assurer des fonctionnalités critiques, les systèmes embarqués imposent des sauts technologiques. Facteurs d'innovation et de différenciation, ils sont essentiels pour le développement des activités industrielles et l'amélioration de la compétitivité. Les systèmes embarqués intègrent une part de plus en plus importante de logiciels « enfouis ». La conception et le développement de tels systèmes sont assujettis à des objectifs économiques, tels que la réduction des coûts et du temps de développement, mais également au respect des normes de la sûreté de fonctionnement. Celui-ci est un des éléments primordiaux et stratégiques. Le fait que ces systèmes soient très présents dans notre environnement quotidien exige des concepteurs de tels systèmes de gérer tout le cycle de développement en garantissant un bon niveau de sûreté de fonctionnement. C’est dans ce contexte que ce chapitre présente les concepts de base et la terminologie de la sûreté de fonctionnement, en insistant sur les spécificités de sûreté des systèmes embarqués. Ensuite, il met en lumière les entraves (défaillances, erreurs et fautes) pouvant se produire dans de tels systèmes ainsi que des exemples de techniques de recours face à de telles entraves, qu’elles soient des techniques de détection uniquement ou de détections et de correction, ou qu’elles soient des techniques de protection logicielle ou des techniques de protection matérielle. Nous

présentons ensuite certains processeurs tolérants aux fautes existants, leurs spécificités et leurs utilités. Nous concluons par leurs propres limites face à notre besoin.

# A. Sûreté de fonctionnement – concepts de base :

## A.1. Introduction – Évolution historique des besoins en terme de sûreté

Avant de donner les nombreuses définitions exactes et de résumer certaines notions, il est bon de rappeler l'historique des évolutions industrielles ou autres qui ont amené à préciser ces notions. Dans les années du milieu du XX$^{ème}$ siècle, il s'est avéré que les produits fabriqués, de technologie de plus en plus complexe, n'avaient pas la fiabilité qu'on pouvait en espérer : nombreux appareils défectueux avant même d'être livrés, fonctionnement ne répondant pas aux besoins, mauvaise adaptation à la maintenance, fragilité, etc. [GIRA04]. Ce sont les notions de complexité d'une part et de recherche d'une plus grande sécurité d'autre part, qui ont influencé l'évolution industrielle au cours du XX$^{ème}$ siècle. En ce qui concerne la « sécurité » et la « sûreté de fonctionnement », les utilisateurs aussi bien que les concepteurs, confrontés aux dysfonctionnements divers, évoquaient la notion de « mauvaise qualité » ou de « fiabilité insuffisante » mais aussi de risques (risques de panne, risque d'indisponibilité, risque d'accident) avec une hiérarchie allant du simple risque de mauvais fonctionnement à la catastrophe. Différents acteurs de développement ou de l'utilisation avaient peu de liens entre eux : des utilisateurs confrontés à des produits mal adaptés ou non fiables, des développeurs peu au courant de l'utilisation réelle des produits, des fabricants confrontés aux défectuosités lors des contrôles, des responsables de sécurité souhaitant réduire le nombre d'accidents. Ainsi, les problèmes rencontrés commençaient à être classés en deux catégories, celles relevant de la :

- sûreté de fonctionnement d'une part, incluant fiabilité (réponse au risque de panne), maintenabilité (réponse au risque de maintenance difficile, voire impossible), disponibilité (réponse au risque de non-mise à disposition au moment du besoin) et,
- sécurité d'autre part, (réponse au risque d'accident ou de catastrophe).

L'ensemble de ces évolutions s'est fait progressivement à partir des années 1950 et 1960. L'électronique et l'informatique de pointe ont été les secteurs pionniers de la sûreté de fonctionnement en raison, entre autres, d'une volonté politique de réussite dans les secteurs spatial ou nucléaire. Le domaine militaire conventionnel a suivi rapidement, puis celui du

civil complexe (aéronautique, centraux téléphoniques, etc.), enfin le domaine grand public (automobiles, téléviseurs, etc.).

En matière de réglementation et de normalisation, la prise en compte des notions de sûreté a amené à constituer des groupes de travail dans les différents organismes nationaux ou internationaux, puis à établir des liens entre eux. L'objectif était de rendre le plus cohérent possible les nombreux textes existants. Cependant, il est important de constater une scission entre les groupes qui incluent la sécurité dans la sûreté de fonctionnement et ceux qui la mettent à part, une difficulté supplémentaire étant apportée par la traduction anglais-français (exemple : *dependability*, *security*, *safety*).

Enfin, un critère qui s'est développé progressivement est celui du facteur humain, dont les dysfonctionnements ajoutent un éclairage supplémentaire aux analyses de risque. Ainsi, au fil des temps, à partir de différents domaines de l'activité industrielle et de différentes fonctions dans l'entreprise, les acteurs économiques ont regroupé l'ensemble des problèmes liés aux risques dans ces deux notions de sûreté de fonctionnement et de sécurité, qui se retrouvent dans la notion très globale de maîtrise des risques [GIRA04].

Après cette brève synthèse de l'évolution historique qui a amené à l'étude et à la détermination des notions de sûreté de fonctionnement et de sécurité, nous donnons dans la suite les diverses définitions des fonctions, comportements et services d'un système ainsi que les concepts de base de la sûreté. Nous présentons ensuite les attributs de la sûreté (disponibilité, fiabilité, sécurité-innocuité, confidentialité, intégrité, maintenabilité), les entraves de la sûreté (fautes, erreurs, défaillances) et enfin les moyens pour la sûreté (prévention des fautes, tolérance aux fautes, élimination des fautes, prévision des fautes).

## A.2. Définitions :

Dans [AVIZ00] et [AVIZ04], la **sûreté de fonctionnement** est définie comme l'aptitude à délivrer un service de confiance justifiée. Cette définition met l'accent sur la justification de la confiance, cette dernière pouvant être définie comme une dépendance acceptée explicitement ou implicitement. La **dépendance** d'un système d'un autre système est l'influence, réelle ou potentielle, de la sûreté de fonctionnement de ce dernier sur la sûreté de fonctionnement du système considéré. La **fonction** d'un système est ce à quoi le système est destiné, comme elle est décrite par la spécification fonctionnelle, qui inclut les performances attendues du système. Le **comportement** d'un système est ce que le système fait pour accomplir sa fonction, et est décrit par une séquence d'états. Le **service** délivré par un

système est son comportement tel que perçu par son ou ses utilisateurs. Un **utilisateur** est un autre système, éventuellement humain, qui est en interaction avec le système considéré. La partie de la frontière du système où ont lieu les interactions avec ses utilisateurs est l'**interface** du service. Un service est considéré correct si et seulement si le service délivré accomplit la fonction du système. La **défaillance** (du service) est un événement qui survient lorsque le service délivré dévie du service correct, soit parce qu'il n'est plus conforme à la spécification, soit parce que la spécification ne décrit pas de manière adéquate la fonction du système. Une **erreur** est une partie de l'état susceptible d'entraîner une défaillance. Une **faute** est une cause adjugée ou supposée d'une erreur. Les modes de défaillance sont les manières selon lesquelles un système peut défaillir, classées selon leur gravité [LAPR04].

La **sûreté de fonctionnement** peut aussi être définie comme l'aptitude à éviter des défaillances du service plus fréquentes ou plus graves que ce qui est acceptable. Les défaillances du service plus fréquentes ou plus graves que l'acceptable sont les défaillances de la sûreté de fonctionnement.

## A.3. L'arbre de la sûreté de fonctionnement :

La décomposition systématique des concepts de la sûreté de fonctionnement consiste en trois parties : attributs, moyens et entraves par lesquelles la sûreté est atteinte [LAPR95] [LAPR04]. Cette décomposition est schématisée par l'arbre suivant (Figure 1.1) :

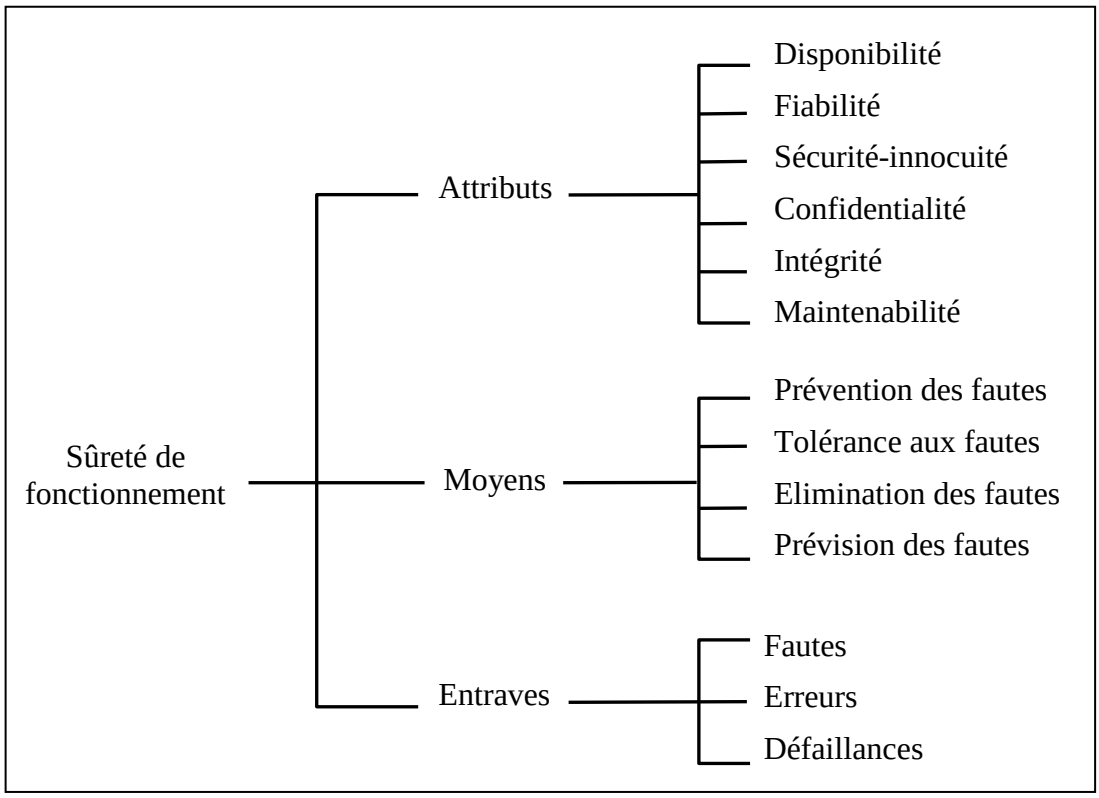


**Figure 1.1 : L'arbre de la sûreté de fonctionnement**

Définissons dans les paragraphes suivant l'ensemble des éléments constituant cet arbre.

### A.3.1. Attributs de la sûreté de fonctionnement

Selon la ou les applications auxquelles le système est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement, ce qui revient à dire que la sûreté de fonctionnement peut être vue selon les propriétés différentes, mais complémentaires, qui permettent de définir ses attributs :

- le fait d'être prêt à l'utilisation conduit à la **disponibilité** (*availability*) ;
- l'absence de conséquences catastrophiques pour l'environnement conduit à la **sécurité-innocuité** (*safety*) ;
- l'absence de divulgations non-autorisées de l'information conduit à la **confidentialité** (*confidentiality*) ;
- l'absence d'altérations inappropriées de l'information conduit à l'**intégrité** (*integrity*) ;
- l'aptitude aux réparations et aux évolutions conduit à la **maintenabilité** (*maintenability*).

Ces six attributs sont les attributs primaires. Il existe également des attributs secondaires, à savoir :

- **robustesse :** persistance de la sûreté de fonctionnement en présence de fautes externes ;
- **survivabilité :** persistance de la sûreté de fonctionnement en présence de fautes actives ;
- **résilience :** persistance de la sûreté de fonctionnement en face de changements fonctionnels, environnementaux, technologiques ;
- **responsabilité :** disponibilité et intégrité de la personne qui a effectué une opération ;
- **authenticité :** intégrité du contenu et de l'origine d'un message, et éventuellement d'autres informations, comme l'instant d'émission ;
- **non-réfutabilité :** disponibilité et intégrité de l'identité de l'émetteur d'un message (non-réfutation de l'origine), ou du destinataire (non-réfutation de la destination).

### A.3.2. Moyens pour la sûreté de fonctionnement

Le développement d'un système sûr de fonctionnement passe par l'utilisation combinée d'un ensemble de méthodes qui peuvent être classées en :

- **prévention des fautes :** comment empêcher l'occurrence ou l'introduction de fautes ;
- **tolérance aux fautes :** comment fournir un service à même de remplir la fonction du système en dépit des fautes ;
- **élimination des fautes :** comment réduire la présence (nombre, sévérité) des fautes ;
- **prévision des fautes :** comment estimer la présence, le taux futur, et les possibles conséquences des fautes.

Nous donnerons dans le paragraphe B de ce chapitre des exemples de moyens pour la sûreté de fonctionnement à savoir, les moyens de recours face aux entraves. Ces moyens peuvent être des techniques de détection ou des techniques de détection et correction à la fois comme ils peuvent être des techniques logicielles ou matérielles. Pour cela, donnons les définitions des différentes entraves à la sûreté de fonctionnement.

### A.3.3. Entraves à la sûreté de fonctionnement

Le service délivré étant une séquence d'états externes, une défaillance du service signifie qu'au moins un état externe dévie du service correct. La déviation est une **erreur**. La cause adjugée ou supposée d'une erreur est une **faute**. Les fautes peuvent être internes ou externes au système. La présence antérieure d'une vulnérabilité, c'est-à-dire d'une faute interne qui permet à une faute externe de causer des dommages au système, est nécessaire pour qu'une faute externe entraîne une erreur et, éventuellement, une **défaillance**. Généralement, une faute cause d'abord une erreur dans l'état interne d'un composant, l'état externe du système n'étant pas immédiatement affecté. Il s'ensuit la définition d'une **erreur** : partie de l'état total du système qui est susceptible d'entraîner sa défaillance, qui survient lorsque l'erreur affecte le service délivré à l'utilisateur. Il est à noter que nombre d'erreurs n'affectent pas l'état externe du système, et donc ne causent pas de défaillance.

La spécification de sûreté de fonctionnement d'un système décrit ce qui est requis pour les attributs de la sûreté de fonctionnement en termes de fréquence et de gravité des défaillances du service pour un ensemble donné de fautes, pour un environnement opérationnel donné. Ainsi, il s'ensuit une définition alternative de la sûreté de fonctionnement, donnée au début de ce chapitre, et qui complète la définition initiale en procurant un critère pour décider si la confiance dans le service peut être placée ou non : aptitude à éviter des défaillances du service plus fréquentes ou plus graves que l'acceptable. Des défaillances du service plus fréquentes ou plus graves que ce qui est acceptable manifestent une **défaillance** de la sûreté de fonctionnement.

Ainsi, nous pouvons résumer ces définitions en ces quelques mots : une **erreur** est un état susceptible d'entraîner une défaillance ; cette erreur peut être latente comme elle peut être détectée ; la propagation d'une erreur peut produire d'autres erreurs ; une **faute** est une cause adjugée ou supposée d'une erreur ; une **défaillance** est une manifestation d'une erreur qui par propagation traverse la frontière du système avec son environnement. Nous pouvons ainsi schématiser la relation entre ces entraves par la figure suivante :

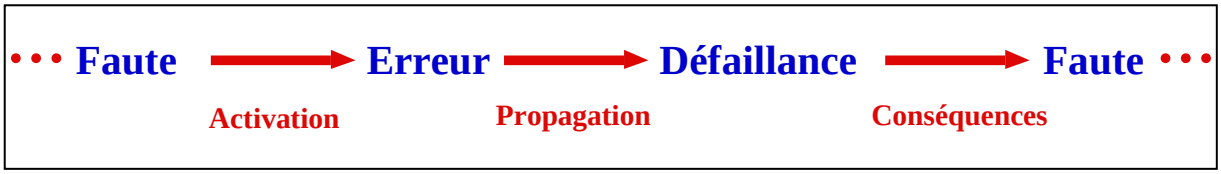


**Figure 1.2 : La chaîne fondamentale des entraves à la sûreté de fonctionnement**

Les flèches de cette chaîne expriment la relation causale entre fautes, erreurs et défaillances. Elles doivent être interprétées de façon générique : par propagation, plusieurs erreurs sont généralement générées avant qu'une défaillance ne survienne.

Maintenant que ces notions sont définies, nous présentons dans la suite les principaux types d'erreurs susceptibles de se produire dans les systèmes embarqués dans un premier temps. Dans un second temps, nous donnons une synthèse des principaux moyens de recours à de telles entraves : les moyens pour la sûreté.

# B. Synthèse des entraves et des moyens de la sûreté de fonctionnement

Un système mécatronique est globalement composé de l'ensemble {processeurs, capteurs, actionneurs et application}. Afin que la sûreté globale de ce système soit atteinte, la sûreté des éléments le constituant ainsi que celle des interfaces et des lignes de transmission les reliant doit être prise en compte dès le début de la phase de conception [RIEM01]. Ceci par une identification des entraves susceptibles de se produire dans de tels systèmes en premier lieu et en second lieu, en y intégrant des moyens pour la sûreté qui sont utilisés en tant que recours à de telles entraves. Nous donnons ainsi dans cette partie une synthèse des entraves et des moyens de la sûreté de fonctionnement.

## B.1. Entraves

Les effets singuliers, regroupés sous le nom **SEE** (*Single Event Effect* : effet d'une particule isolée), correspondent aux phénomènes déclenchés par le passage d'une particule unique, telles que les ions lourds ou des protons énergétiques. Ces effets singuliers sont classés en deux sous-catégories [FAUR05], [SADA05] :

- Les effets irréversibles, dégradations permanentes destructifs appelés erreurs permanentes ou erreurs matérielles. Parmi ces effets, nous citons le **SEL** (*Single Event Latchup* : *latchup* par une particule isolée).
- Les effets réversibles, défauts transitoires non destructifs, appelés aussi aléas logiques ou erreurs logicielles. Parmi ces effets, nous citons le **SET** (*Single Event Transient* : transition par une particule isolée) et le **SEU** (*Single Event Upset* : perturbation par une particule isolée)

En effet, parmi les aspects essentiels qui intéressent le concepteur d'architecture de processeur, c'est la susceptibilité de cette architecture envers les SEEs. L'effet d'une particule isolée SEE est par définition la modification du fonctionnement de composants électroniques, causée de façon aléatoire par une particule de haute énergie. La particule peut ainsi endommager les composants ou perturber l'information qu'ils fournissent [CGTN05] [ACTEL] [AERO]. Parmi les SEEs les plus répandus, nous trouvons le *latchup* par une particule isolée (SEL : *Single Event Latchup*), qui se produit lors de la création de transistors parasites à cause d'une fausse pointe de courant (*spurious current spike*) ; Ce qui implique un court-circuit jusqu'à claquage ou rafraîchissement de l'alimentation. Nous trouvons également la transition par une particule isolée (SET : *Single Event Transient*), phénomène de propagation transitoire qui peut affecter par exemple la sortie d'une porte logique. Enfin, nous trouvons la perturbation par une particule isolée (SEU : *Single Event Upset*), qui a pour conséquence de modifier l'information fournie par un composant électronique sous l'effet d'une particule de haute énergie [KATZ01]. C'est un basculement de bit qui se produit dans une bascule ou dans un point mémoire [FAUR05].

La famille des évènements singuliers regroupe de nombreux autres. Certains sont destructifs et d'autres non. Nous décrivons dans la suite quelques uns [THOU07], [FAU05].

Parmi les erreurs temporaires, nous citons :

— **MBU** (*Multi Bit Upset*) : analogue aux SEUs mais plusieurs bits sont corrompus par la même particule. Elle affecte plusieurs structures voisines de type bascule en y induisant des changements d'états. La probabilité de ce type d'évènement singulier est plus faible que celle du SEU. Elle croît avec la miniaturisation de l'électronique.

Quant aux erreurs permanentes, nous citons :

— **SEFI** (*Single Event Functional Interrupt*) : les mécanismes conduisant à cet état sont nombreux, et dépendent du circuit cible. Il se caractérise par un comportement erratique du circuit, souvent associé à une augmentation de la consommation. La récupération de l'état normal du circuit passe souvent par un reset complet, voir par un redémarrage de l'alimentation. L'effet singulier se traduit par le blocage de la fonctionnalité du composant en le faisant passer dans des états interdits.

— **SHE** (*Single Hard Error*) : ce phénomène a pour symptôme un bit collé, qu'il est impossible de reprogrammer.

— **SEGR** (*Single Event Gate Rupture*) : le dépôt de charge conduit à la destruction d'une structure MOS par claquage de l'oxyde de grille.

— **SEB** (*Single Event Burn-out*) : ce phénomène s'observe dans les MOSFETs de puissance. Si le transistor est à l'état bloqué et que le courant transitoire est capable de rendre passant le transistor bipolaire parasité, une grande quantité de courant résultante peut détruire le composant par effet thermique.

Nous avons donné un ensemble d'erreurs susceptibles d'attaquer les éléments de mémoires, à savoir les « effets d'une particule isolée » (SEE : *Single Event Effect*). Dans la suite de la thèse, nous nous intéressons aux erreurs temporaires, à savoir les erreurs de type SEU (*Single Event Upset*) et/ou MBU (*Multi Bit Upset*) dans le cas de multiples conséquences sur divers bits. En effet, la manière la plus simple de représenter un SEU est le basculement d'un bit mémoire d'un niveau logique vers son complémentaire. C'est un effet induit par le passage d'une particule ionisante dans ou proche d'un des nœuds sensibles d'un élément de mémorisation. Sans schéma de protection adéquat, capable au minimum de signaler la corruption (utilisation d'un code de parité par exemple), voir de la corriger (code de détection et de correction d'erreur de type Hamming par exemple) il va de soi que l'utilisation d'un bit dont la valeur est corrompue peut avoir de sérieuses conséquences sur le comportement d'un système électronique. Dans ce cas de figure, des mécanismes de protection s'avèrent nécessaires pour remédier à ces problèmes. Le paragraphe suivant fera alors l'objet d'une synthèse des moyens pour la sûreté. Nous présentons également des applications industrielles et/ou pédagogiques intégrant certains de ces moyens appelés également dans la suite de la thèse mécanismes, méthodes ou techniques de protection.

## B.2. Moyens pour la sûreté – Techniques de protection

Comme nous l'avions cité précédemment, il existe quatre moyens pour la sûreté de fonctionnement [LAPR04] : la prévention des fautes, la tolérance aux fautes, l'élimination des fautes et la prévision des fautes, comme le montre la figure 1.3.

La tolérance aux fautes consiste en la détection d'erreur et le rétablissement. Le rétablissement est effectué en traitant les erreurs ou en traitant les fautes. Traiter les erreurs consiste à détecter l'existence d'un état incorrect (erreur) et à remplacer l'état incorrect par un état correct (conforme aux spécifications). Traiter les fautes consiste à prévoir des composants multiples pour réduire la probabilité qu'une faute conduise à une défaillance et à réaliser des traitements multiples, par exemple réaliser la même opération par des algorithmes différents

pour tenter d'éviter les fautes de conception. L'élimination des fautes peut se faire soit pendant la phase de développement comme elle peut également se faire pendant la phase de la vie opérationnelle du système. Enfin, pour la prévision des fautes, deux cas de figures sont possibles. L'évaluation ordinale ou qualitative à travers une analyse des modes de défaillances, à travers les diagrammes de fiabilité ou aussi les arbres de fautes. L'évaluation probabiliste ou quantitative à travers des modélisations tels que les diagrammes de fiabilité, les chaînes de Markov ou les réseaux de Pétri stochastiques ou également à travers des tests opérationnels. La prévention des fautes n'est pas détaillée car elle relève de l'ingénierie générale des systèmes et déborde donc largement le cadre de la sûreté de fonctionnement [LAPR04].

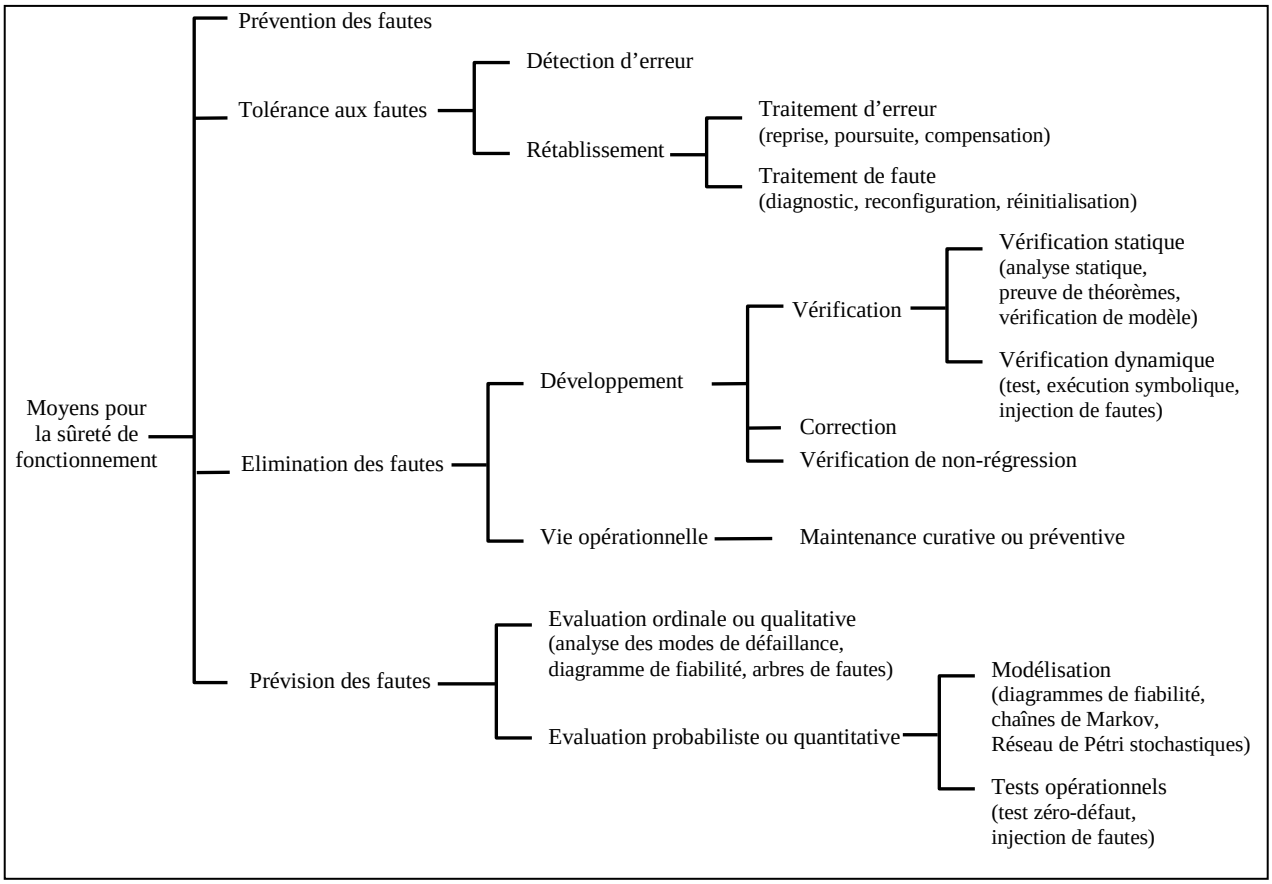


**Figure 1.3 : Moyens pour la sûreté de fonctionnement**

Nous venons de présenter les divers types de moyens pour la sûreté de fonctionnement. Dans la suite, nous donnons une synthèse des principales méthodes de protection utilisées dans le monde industriel ainsi que dans le domaine de la recherche.

Un moyen de faire face aux risques des SEUs consiste à minimiser la probabilité d'apparition des effets qui leurs sont dus. Compte tenu de la gravité des effets des ions lourds sur l'électronique, des moyens d'action face aux radiations s'avèrent nécessaires. De

nombreuses techniques ont été développées pour limiter la sensibilité des composants aux différents effets des radiations ou atténuer l'impact des défaillances sur le fonctionnement global d'un système. Des circuits de surveillances contrôlant le courant d'alimentation, ou le flux de données d'un processeur (*watchdogs*) peuvent être utilisés. Des codes embarqués de détection et de correction d'erreurs (EDAC : *Error Detection And Correction code*, ou ECC : *Error Correction Code*) permettent dans une certaine mesure, de rétablir des données corrompus [MERA07]. Nous détaillons certaines techniques dans la suite.

La tolérance aux fautes peut être réalisée par l'emploi de redondances spatiales (duplication de composants), temporelles (traitements multiples) et informationnelles (codes et signatures).

Il n'existe pas de méthodes de tolérance aux fautes valables dans l'absolu, seulement des méthodes adaptées à des hypothèses particulières d'occurrence de fautes. Ces hypothèses doivent donc être explicitement formulées, après une analyse soigneuse. Parmi les techniques de base de la tolérance aux fautes, nous citons :

### *Le recouvrement* (error recovery)

C'est le fait de remplacer l'état d'erreur par un état correct. Il nécessite une détection de l'erreur (identification de la partie incorrecte de l'état), au moyen d'un test de vraisemblance. Une telle détection est soit explicite (en exprimant et ensuite vérifiant des propriétés spécifiques de l'état), soit implicite (via des conséquences visibles telle que la violation d'un délai de garde ou la violation de protection de mémoire, etc.). Parmi les techniques de recouvrement, nous trouvons la reprise et la poursuite. Concernant la reprise, c'est une technique générale qui consiste en un retour en arrière vers un état antérieur dont on sait qu'il est correct. Elle nécessite donc la sauvegarde de l'état. Elle consiste à choisir des points de reprise, pendant lesquels nous faisons une sauvegarde périodique de l'état du système en utilisant des mécanismes de mémorisation qui protègent les informations, vis à vis des effets des erreurs tolérées et en tenant compte des problèmes de cohérence de l'état global sauvegardé. Ainsi, nous déterminons l'état global cohérent le plus proche et nous reprenons les traitements. Quant à la poursuite, c'est une technique spécifique qui dépend de l'application. C'est une reprise à partir d'un état cohérent futur. Elle consiste en une tentative de reconstitution d'un état correct sans retour arrière. La reconstitution est souvent seulement partielle, d'où le service risque d'être dégradé.

### *La compensation* (error masking)

C’est le cas ou l’état possède une redondance interne suffisante pour détecter et corriger l’erreur. Les tests externes de vraisemblance sont ainsi inutiles. La redondance est utilisée de manière à ce que le système continue à fonctionner correctement même en présence d’erreur. Comme exemple, nous pouvons citer l’application des codes détecteurs et correcteurs ainsi que le vote majoritaire.

### *La détection*

Les objectifs consistent à prévenir, si possible, l’occurrence d’une défaillance provoquée par l’erreur, à éviter la propagation de l’erreur à d’autres composants et à faciliter l’identification ultérieure de la faute en vue de son élimination ou de sa prévention. Les paramètres de la détection sont la latence et le taux de couverture. La latence est le délai entre production et détection de l’erreur. Le taux de couverture est le pourcentage d’erreurs détectées. Si la technique de détection se base sur le principe de comparaison des résultats de composants dupliqués, son coût est élevé, sa latence est faible et son taux de couverture est élevé. Si par contre elle se base sur le principe de contrôle de vraisemblance, son coût est modéré, sa latence est variable selon la technique et son taux de couverture souvent faible.

Deux grands types de techniques sont utilisés : le codage et le vote. Le principe général du codage (binaire) est le suivant : soit `I` l'ensemble des données à coder, `I` est injecté dans un ensemble `C` (le code). La distance du code est définie par `d`, qui est le nombre de bits distincts entre deux mots du code (distance de Hamming). Supposons que `d=2t+1`, alors le code détecte `2t` erreurs binaires et corrige `t` erreurs binaires. Les principaux codes utilisés sont les codes de Hamming (pour les mémoires), les codes polynomiaux (pour les mémoires secondaires et la transmission des données) et enfin les codes arithmétiques (pour la détection des erreurs d'exécution). Quant au vote, plusieurs versions d'une même donnée sont calculées indépendamment. Une erreur est détectée lorsqu'une des versions au moins est distincte des autres. Les unités redondantes doivent être indépendantes vis à vis de la création et de l’activation de la faute. Ils peuvent être soit des copies, soit des éléments diversifiés. Le vote est réalisé par comparaison dans le cas de copies ou par fonction de décision dans le cas d’éléments diversifiés.

La détection peut aussi se faire à travers des contrôles temporels tels que le contrôle du séquencement des instructions ou le contrôle des séquences d’événements, à travers des contrôles de vraisemblance, à travers des contrôles d’intégrité portant sur la structure des

données ou à travers des contrôles de signature : détection des erreurs sur le séquencement et sur la sélection des opérandes d'une opération par exemple.

### *La redondance*

Dans [ROUC92], parmi les divers types de redondance, nous distinguons :

- la redondance active qui est une redondance telle que tous les moyens d'accomplir une fonction requise fonctionnent simultanément (cette redondance permet d'assurer la sécurité) ;
- la redondance passive qui est une redondance telle qu'une partie seulement des moyens d'accomplir une fonction requise est en fonctionnement, le reste n'étant utilisé sur sollicitation qu'en cas de défaillance de la partie en fonctionnement (cette redondance permet d'assurer la disponibilité) ;
- la redondance majoritaire m/n qui est une redondance telle qu'une fonction requise n'est assurée que si au moins `m` des `n` moyens existants sont en état de fonctionner ou en fonctionnement (cette redondance permet d'assurer la sécurité et la disponibilité).

Nous trouvons également d'autres types de redondances telles que la redondance temporelle et la redondance spatiale. Pour un composant soumis à des pannes, il est de pratique courante de tenter de masquer cette panne par un nombre fixé de tentatives successives, ce qu'on appelle redondance temporelle. Lorsque cette stratégie réussit, la défaillance du composant est masquée au niveau supérieur. En cas d'échec une exception est transmise. Comme exemples, nous pouvons citer la reprise dans les protocoles de communication ou la reprise dans les opérations de transfert vers une mémoire secondaire. Dans les deux cas la détection est faite par codage. Quant à la redondance spatiale, un groupe de composants redondants `g` peut-être conçu de telle façon qu'en dépit de la panne de certains membres de `g`, le service offert du point de vue global par les membres de `g` continue, avec éventuellement des performances dégradées. À noter que la redondance ne vaut que si les conditions de défaillance sont indépendantes pour les deux unités redondantes.

Dans d'autres cas, les effets des SEUs peuvent être évités en employant la redondance modulaire duale DMR (*Dual Modular Redundancy*) ; Par exemple, en ajoutant deux bascules plus une porte `ET` à la sortie. Cependant, il y a le problème du risque de glissement temporel à la sortie de la porte `ET`. Ce qui nous amène à parler de la redondance triple modulaire TMR (*Triple Modular Redundancy*). C'est une technique d'implantation de registres : chaque

registre est suivi de 3 bascules ou *latches* de telle manière que nous pouvons déterminer par vote l'état du registre. Cette technique peut être applicable au circuit complet comme à une partie du circuit. Elle peut être directement intégrée dans le code VHDL comme c'est le cas du processeur LEON-FT, architecture SPARC développé par l'agence spatiale européenne et dont nous parlerons plus en détail dans la suite. Il y a également certains outils de synthèse, tels que *Synplify* qui consiste à remplacer chaque bascule par son correspondant en bascules-TMR. Dans ce cas, il n'y a pas de mise à jour du code VHDL à faire par l'utilisateur. Certains FPGA (famille SX-S, ACTEL) incluent des bascules-TMR directement sur silicium. Grâce à la simplicité de l'implantation de la technique du TMR, il est recommandé de l'appliquer dans les sous-circuits critiques incluant les chemins de données, en particulier les registres d'état et les registres de configuration, surtout dans les applications ou le rafraîchissement de l'alimentation ou le reset est impossible. Cependant, l'utilisation de cette technique présente quelques problèmes. En effet, une erreur SEU ne peut pas être évitée si une seconde erreur apparaît avant le cycle de rafraîchissement des bascules. Dans ce cas, il faudra corriger la première erreur avant l'apparition de la seconde et par conséquent, utiliser des bascules dont la fréquence de rafraîchissement est supérieure à celle de l'apparition de l'erreur. Dans la figure suivante, nous avons une protection de la mémorisation de l'entrée.

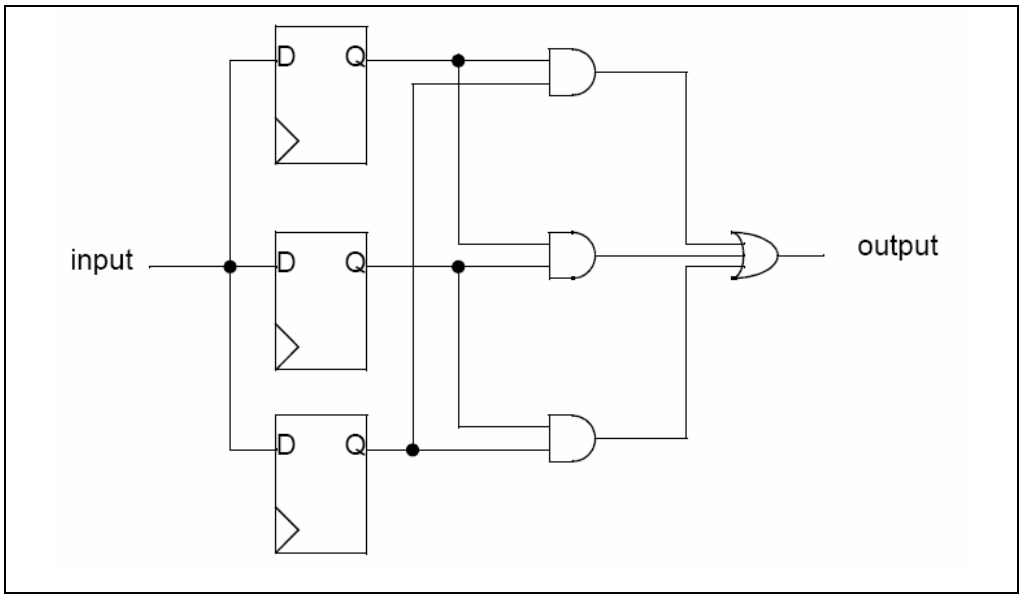


**Figure 1.4 : Protection d'un point mémoire par la technique de TMR**

Nous pouvons également trouver une seconde manière, telle qu'elle est illustrée dans la Figure 1.5. Dans ce schéma, nous exploitons les ressources nécessaires présentes dans le FPGA, à savoir des multiplexeurs au lieu des portes `ET` et `OU` comme c'est le cas dans la Figure 1.4.

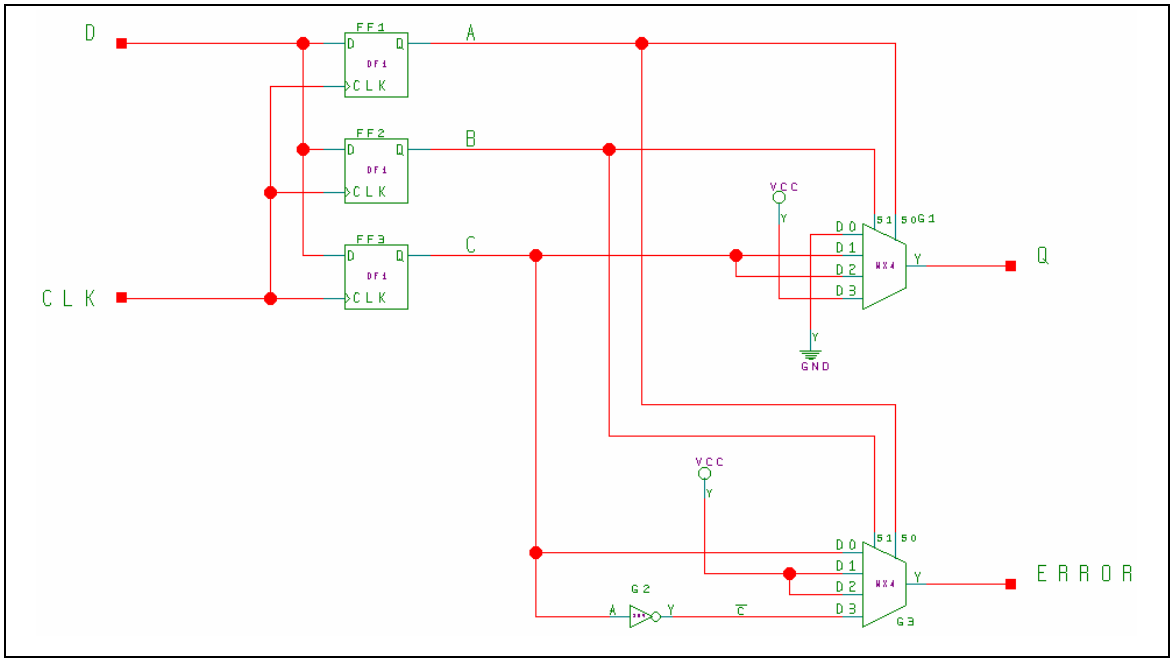


**Figure 1.5 : Autre circuit de protection d'un point mémoire par la technique TMR**

Une seconde sortie est ajoutée (sortie *ERROR*), pour détecter s'il y a un désaccord entre les sorties des trois bascules. Nous pouvons nous en apercevoir plus clairement grâce à la table de vérité suivante :

| C | B | A | Q | Error |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 = D0 | 0 (C) |
| 0 | 0 | 1 | D1=C=0 | 1 |
| 0 | 1 | 0 | D2=C=0 | 1 |
| 0 | 1 | 1 | 1 = D3 | 1 (C) |
| 1 | 0 | 0 | 0 = D0 | 1 (C) |
| 1 | 0 | 1 | D1=C=1 | 1 |
| 1 | 1 | 0 | D2=C=1 | 1 |
| 1 | 1 | 1 | 1 = D3 | 0 (C) |

Une des trois entrées est différente

**Tableau 1.1 : Table de vérité de la Figure 1.5**

### *Codes détecteurs d'erreurs (CDE) [PIES92]*

L'utilisation des codes détecteurs d'erreurs (CDE) ne peut se justifier que si elle conduit à une redondance globale inférieure à celle entraînée par la duplication du module fonctionnel suivie d'un comparateur. Il faut donc que le *hardcore* (le matériel dont le bon fonctionnement est nécessaire pour réaliser les fonctions de diagnostic, comme le comparateur) soit lui-même périodiquement testable hors ligne, ou mieux, autotestable en ligne. Le schéma synoptique

d'un circuit fonctionnel TSC (*Totally Self Checking*) est représenté Figure 1.6 ; il montre les places respectives des codes détecteurs d'erreurs sur les `n` entrées et les `s` sorties du circuit fonctionnel, ainsi que celles des contrôleurs autotestables STC (*Self Testing Checkers*) délivrant si besoin les signaux d'erreur. Ces contrôleurs doivent avoir au moins deux sorties complémentaires codées, pour ne pas être tributaires d'un collage interne (à `1` ou à `0`), tel qu'à l'entrée on ait : soit `00`, soit `11` [GIRA07].

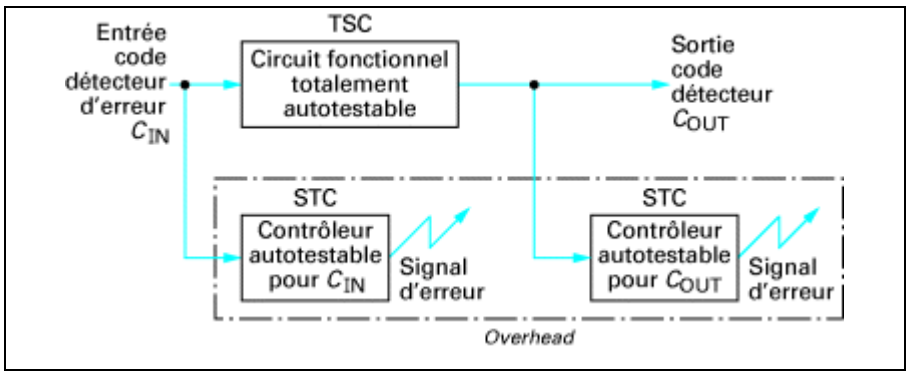


**Figure 1.6 : Synoptique générale d'un système totalement autotestable (TSC)**

En outre, nous rappelons les définitions suivantes :

- **Codes systématiques** : codes où les bits d'information peuvent être distingués des bits de contrôle (par leur place) ;
- **Codes séparables** : codes systématiques dont les parties information et contrôle de l'opérande sont **traitables séparément** (le décodage ignore le(s) octet(s) de contrôle) ;
- **Notion de couverture** : soient *X* et *Y* deux suites binaires de longueur *n*, on dit que *X* couvre *Y* (noté $Y \leq X$) si et seulement si *X* comporte des « 1 » partout où *Y* en a. Si, par contre, **ni** $Y \leq X$, **ni** $X \leq Y$, *X* et *Y* sont alors **non ordonnées**.

### *Codes correcteur d'erreurs*

Nous nous contentons ici à une présentation sommaire de **codes linéaires à blocs**, d'implémentation aisée, sans approfondissements théoriques [BERL68], [McEL77], dans la mesure où notre objectif dans ce chapitre est de donner une synthèse brève entre autres des moyens pour la sûreté de fonctionnement.

Dans la suite, nous appelons :

- **Mot :** un regroupement de bits, dont la longueur – fixe – est le nombre de bits.

— **Symbole :** un regroupement de bits **ayant un sens**, dans l'opération (codage, modulation, etc.) en cours ; un symbole n'étant pas nécessairement de longueur fixe.

— **Bloc :** le regroupement de symboles/mots, lié au codage détecteur/correcteur d'erreurs.

Le schéma d'une liaison met en jeu : une source numérique du **message émis $M_E$**, un émetteur du signal modulé correspondant **$S_E$**, un canal de transmission par lequel parvient un signal **$S_R$** au récepteur, qui restitue un message **$M_R$** au destinataire. Entre **$M_E$** et la modulation générant **$S_E$**, l'émetteur effectue en fait deux codages : un codage source destiné à supprimer toute redondance inefficace dans la sémantique de l'information transmise et un codage canal destiné à y introduire une redondance efficace (permettant la détection/correction d'erreurs) **;** Bien évidemment, il existe les opérations miroir de décodage dans le récepteur. Seul les codage/décodage **canal** nous intéressent ici [GIRA07].

Pour pouvoir décréter que les données reçues sont correctes, il faut que celles-ci respectent un critère (le code) que les données erronées ne respectent pas. Le codage consiste donc à faire rentrer les données dans le code, en y calculant des bits que l'on rajoute à un groupe de données pour faire un **bloc** du code. Comme la transmission peut modifier ce bloc, le décodage consistera à vérifier, **par le calcul du syndrome `s(*)`**, si le bloc appartient au code.

Si ce n'est pas le cas (**`s¹0`**), le calcul du syndrome détectera s'il y a une (des) erreur(s) et permettra leur localisation et correction éventuelle.

On démontre que pour corriger **`t`** erreurs d'un code, il faut que les blocs utilisés au décodage soient à **une distance minimale de (`2t+1`), au sens de Hamming,** c'est-à-dire que le nombre de bits différents, entre chacun des mots du code soit (**`2t+1`**) [BERL68], [McEL77]. Un paramètre fondamental, relativement à la probabilité d'erreur attendue au décodage, est le **taux `R` d'information des mots transmis, par unité de temps**. En effet, supposons par exemple que **`R = 1/3`**, cela veut dire que le canal peut transmettre des bits trois fois plus vite que la source ne les produit, donc que la sortie de la source peut être encodée avant la transmission. On conçoit donc qu'il est possible, si **`R`** est très petit, de rendre cette probabilité très faible aussi.

Un des modes le plus trivial est **le codage par réplication**. Si à l'émission, nous avons répété deux fois l'information dans le bloc et qu'à la réception, l'information et son duplicata

sont identiques, nous admettons qu'elles n'ont pas été perturbées ; sinon il faut tripler l'information pour pouvoir corriger par vote majoritaire. Ce codage est très consommateur en ressource (rendement 1/3), mais ne présente pas de contrainte sur la taille du bloc ; il est facile à implémenter et est encore utilisé pour les parties très sensibles de message, car il peut corriger plusieurs erreurs consécutives.

Nous trouvons également **les codes à simple contrôle de parité**. Le contrôle de parité peut être horizontal/vertical et à parité paire ou impaire. Les figures Figure 1.7 et Figure 1.8 montrent respectivement l'organisation d'un UART (*Universal Asynchronous Receiver Transmitter*) combinant un bit de parité à la transmission série et l'élaboration d'un bit du vérificateur de parité verticale.

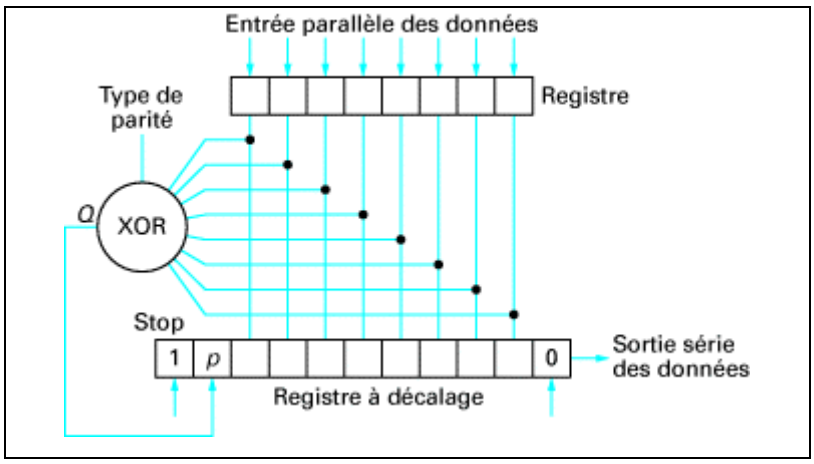


**Figure 1.7 : Génération du bit de parité d'une transmission série**

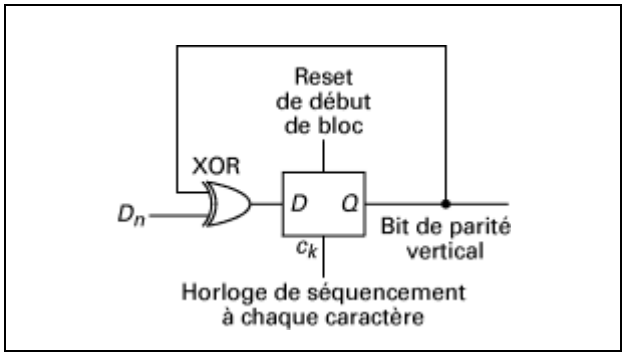


**Figure 1.8 : Élaboration du bit de parité vertical**

Là encore le principe est très consommateur en ressources, mais facile à implémenter et toujours utile pour la sécurisation de messages asynchrones. Toutefois, nous notons que la parité verticale demande beaucoup de mémoire et qu'il faut **à la fois** disposer de parités horizontale et verticale pour faire de la **correction d'erreur simple**. Précisons enfin que l'opération mathématique de codage, n'est pas tolérante aux fautes [GIRA07].

Concernant le code de Hamming H (B, D) [HAMM50], nous notons B la longueur de chaque bloc de code en bits et D le nombre de bits d'information dans un bloc. R = D / B est le taux d'information du bloc. Soit m le nombre de bits de parité du bloc :

- Pour corriger une erreur : $B = 2^m - 1$ et $D = 2^m - 1 - m$,
  d'où $R = 1 - m\,(2^m - 1)^{-1}$ ;
- Pour corriger une erreur et détecter deux erreurs : il faut rajouter un bit de parité à l'ensemble alors $B' = 2^m$, mais le nombre de bits d'information reste le même.

Les codes de Hamming généralisent astucieusement la technique du bit de parité, comme le montre explicitement sur la Figure 1.9 la génération des équations de vérification de la parité pour un code H (15, 11). Cette technique **linéaire** de codage (**où la somme modulo 2 de deux mots de code est toujours un mot de code**) s'est avérée idéale pour un fond de panier et la correction en **parallèle** de petits messages.

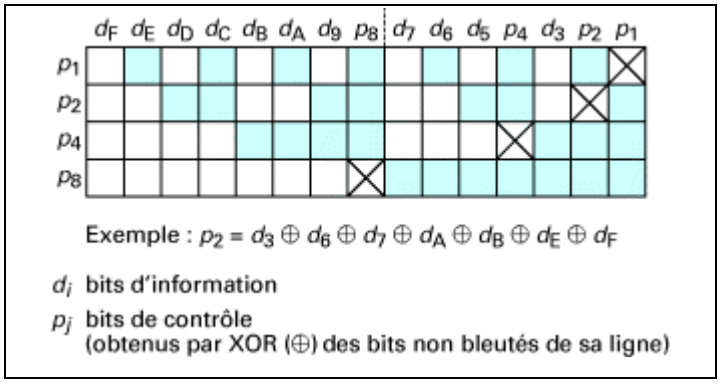


**Figure 1.9 : Élaboration du bit de parité vertical**

Dans la correction d'erreurs multiples, on conçoit que les codes soient plus redondants et le décodeur plus complexe que ceux d'Hamming pour une correction unique. Ce sont les codes CRC (*Cyclic Redundancy Code*) [GIRA07]. Comme par ailleurs le codage et le décodage d'informations nécessitent l'exécution d'opérations sur leurs représentations binaires, seules les techniques alliant un bon modèle mathématique à une implantation électronique performante (rapide et peu exigeante en ressources, donc fiable) se sont imposées. Notamment en mathématique : polynômes formels, arithmétique modulaire, corps de Galois, polynômes générateurs ; Et en électronique : transformation série-parallèle, registre à décalage, opérateurs élémentaires, filtres numériques. L'information numérisée sous forme d'un train binaire peut être représentée comme une suite $\{a_0, a_1, \ldots, a_n\}$ et correspond à la structure d'un bus parallèle de largeur *n*. Cette suite figure en général le nombre $a_0 + a_1$ à $2 + \ldots + a_n$ à $2^n$ de l'arithmétique usuelle. Pour réaliser économiquement des opérations sur des trains de bits, on les **sérialise** pour les faire tous passer à travers **un**

**registre à décalage** (figure 1.10), qui n'est autre qu'une succession de points mémoire reliés entre eux. Chaque coup d'horloge fait apparaître le bit suivant à la sortie du registre. On obtient ainsi une bonne représentation par trains de bits sérialisés de polynômes formels de l'indéterminée x (sur Z/2, puisque les coefficients sont binaires). À noter que *x* ne prenant ni la valeur 0, ni la valeur 1, doit ici être interprété ici comme un **délai**.

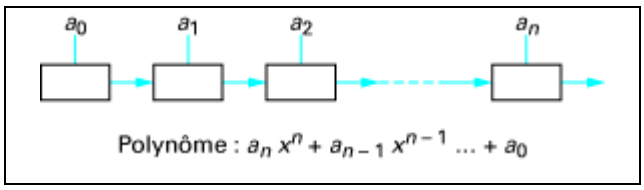


**Figure 1.10 : Élaboration du bit de parité vertical**

Matériellement on peut réaliser n'importe quelle opération sur les booléens, la plus utilisée étant le XOR qui correspond à l'addition modulo 2 des entiers :

a XOR b = 1 si a et b sont différents, 0 sinon

Cette addition a des propriétés mathématiques intéressantes : chaque nombre est son opposé (0 + 0 = 0 et 1 + 1 = 0) donc addition et soustraction sont la même opération. Elle a aussi de bonnes propriétés en termes de circuits : comme il n'y a pas de retenue, il n'y a pas de report d'un bit sur un autre, l'opération peut être faite sans introduire de délai supplémentaire. En combinant les deux opérations précédentes, on a trouvé une architecture générique permettant d'effectuer les opérations sur des trains de bits, comme les multiplications ou divisions – des polynômes qui leur correspondent –, par un polynôme fixe.

Après cette taxonomie de divers moyens pour la sûreté de fonctionnement, nous présentons dans la suite quelques applications de ces moyens dans certaines applications industrielles et/ou pédagogiques.

Certains travaux intègrent les codes de détection et de correction intégrés dans leurs systèmes comme c'est le cas de [CHEN83]. Les correcteurs d'erreur simple et détecteurs d'erreur double capables de détecter toutes les erreurs d'un seul caractère s'avèrent importants pour les applications pratiques. Ceux-ci sont utilisés pour améliorer la fiabilité et l'intégrité des données des systèmes de mémoire des ordinateurs [CHEN83]. [CHEN84] présente l'état de la situation pour les codes correcteurs d'erreurs utilisés dans les applications pour les mémoires à semi-conducteurs pour ordinateurs. Dans [ZARA07], des codes de détection et de corrections sont utilisés en tant que recours face aux SEUs susceptibles de se produire dans les LUTs (*Look-Up Table*) du FPGA. [KOCH04] propose d'inclure une évaluation de la sûreté au cours de toutes les étapes du cycle en V servant pour la conception et le

développement de systèmes mécatroniques. [ISER02] a étudié un système de freinage câblé composé d'une unité opérationnelle (volant et pédale de frein) avec une sortie électrique, un retour au conducteur, des bus, des calculateurs et des actionneurs. Il a inclut l'analyse des modes de défaillances et de leurs effets lors de la phase de conception. Il a également intégré une redondance et un voteur dans chacun des composants de l'électronique. Il a aussi utilisé la redondance pour les capteurs et les actionneurs. [DILG04] montre que dès le début du concept d'une voiture jusqu'à son déploiement, les systèmes mécatroniques présents dans les voitures doivent prendre en charge la possibilité d'une self-réparation, la fiabilité des circuits numériques et analogiques et celle des capteurs et des actionneurs et enfin, le self-diagnostic dans le canal de communication. [DILG03] porte sur les extensions ajoutées à un capteur d'angle de direction utilisé dans un système de freinage câblé en vue d'améliorer la tolérance aux fautes de ce système. Ces extensions consistent d'une part dans l'ajout de deux éléments optiques nouvellement intégrés dans le capteur existant. D'autre part, il a exploité des relations mathématiques données par l'angle de direction. [STOJ01] a intégré un voteur dans son architecture en utilisant le principe de la redondance. Au cas d'une erreur, la valeur erronée est remplacée par la valeur correcte. [SING06] a par contre utilisé une combinaison de quelques techniques pour la tolérance aux fautes et pour la détection à savoir, la duplication, la réplication, les *checkpoint*/*restart*, les *breakpoints* et les TMR.

## C. Processeurs tolérants aux fautes

Nombreux travaux de recherche ont été attribués à la sûreté de fonctionnement et à la tolérance aux fautes en particulier en tant que un des plus importants aspects de la sûreté. Des techniques de protection qu'elles soient logicielles ou matérielles ont été prises en compte. Jusqu'à cette date, quelques processeurs tolérant aux fautes (*Fault-Tolerant Processors*) ont vu le jour. Parmi eux nous pouvons citer le microprocesseur LEON-FT [GAIS02] et le microprocesseur IBM S/390 G5 [SLEG99]. Le LEON-FT utilise la technique du TMR dans chaque bascule. L'IBM S/390 G5 reproduit de larges portions de son cœur, incluant les *I-unit* (*fetch and decode*) et les E-*unit* (*execution and register file*), ce qui ajoute considérablement la surface. D'autres travaux sont orientés vers les techniques de détection de fautes dans des cœurs de processeurs simple (Argus [MEIX08]), en se basant sur la vérification dynamique des quatre tâches performées sur le microprocesseur Von Neumann : le flux de contrôle, le flux de données, le calcul et l'accès mémoire. L'approche de [SHYA06] vise la protection du pipeline du microprocesseur et de la mémoire des défaillances provenant du silicium.

Parmi les processeurs qui incluent la technique TMR, nous trouvons le processeur LEON-FT [GAIS00], [GAIS02], [GAIS03]. C'est un projet qui a débuté fin 1997 par des chercheurs de l'agence spatiale européenne ESA (*European Space Agency*) et le centre de technologie et de recherche spatiale européenne ESTEC (*European Space research and Technology Centre*). Il a été abordé suite au besoin de l'industrie européenne aérospatiale d'un processeur 32-bit tolérant aux radiations et de haute performance d'un coté. De l'autre coté, une des raisons de développement de ce projet était surtout pour être indépendant des processeurs américains qui ont des restrictions d'usage, une difficulté à leur amélioration et une protection du savoir faire.

Le LEON-FT permet entre autres la détection et la tolérance aux fautes dans n'importe quel registre du circuit sans intervention logicielle ainsi que la suppression implicite de l'effet des SEUs dans la logique combinatoire. Tous les registres sont implantés en utilisant la TMR, la modification est faite directement dans le code VHDL et il n y a pas de glissement temporel à la sortie lors de l'occurrence des SEUs.
Le modèle VHDL du processeur 32-bit LEON est conforme à l'architecture SPARC V8. Il est conçu pour des applications embarquées avec les fonctions suivantes directement sur le chip :

- instructions et cache séparés ;
- Multiplicateur et diviseur matériel ;
- Contrôleur d'interruption ;
- Deux temporisateurs de 24-bit ;
- Deux UART ;
- Chien de garde ;
- Port 16 bits d'entrée/sortie ;
- Contrôleur de mémoire flexible.

Des modules additionnels peuvent être ajoutés en utilisant les bus AMBA AHB/APB. Le modèle VHDL est synthétisable avec la majorité des outils de synthèse et peut être implanté sur des FPGAs et ASICs.

### *LEON3FT*

Le processeur LEON3FT est une version tolérante aux fautes du LEON3 (architecture SPARC V8). Il est conçu pour la détection et la correction des erreurs SEUs dans les circuits de mémoires RAM. Le LEON3FT supporte la majorité des fonctionnalités du standard LEON3, en plus il dispose de ces nouvelles caractéristiques :

— Correction de plus que quatre erreurs SEUs dans un registre de 32bits,
— Correction de plus que quatre erreurs SEUs par étiquette ou mot de 32bits dans la cache mémoire,
— Traitement des erreurs autonome et,
— Pas d'impact sur les délais lors de la détection et correction des erreurs.

La tolérance aux fautes dans le LEON3FT est implantée en utilisant les codes ECC (*Error-correction coding, Efficient channel coding*) dans tous les blocs mémoires. Les codes ECC sont adaptés selon la technologie des blocs de mémoires et selon le type des données stockées.

L'objectif est de détecter et corriger plus que quatre erreurs par un mot de 32bits. Dans les blocs RAM ou les données sont dupliquées dans une seconde zone mémoire (exemple : mémoire cache), les codes ECC servent pour la détection uniquement. Dans ce cas, la phase de correction consiste à recharger la donnée erronée à partir de la seconde zone mémoire. Dans les mémoires caches, ceci est équivalent à l'invalidation de la ligne cache (*cache line*) erronée et sa rechargement à partir de la mémoire principale.

Dans les blocs RAM ou il n y a pas possibilité d'une duplication de données dans une seconde zone (exemple : registre – *register file*) les codes ECC servent pour la détection et la correction d'erreurs. On s'intéresse aux temps d'encodage/décodage plutôt que de minimiser le nombre des bits ECC. Cette approche garantit que la logique FT n'affecte pas les performances temporelles du processeur (on atteint la même fréquence maximale que le LEON3 non FT). L'encodage/décodage ECC est fait dans le pipeline du LEON3FT en parallèle avec les opérations normales, et le cycle de correction est totalement transparent au logiciel sans effet sur les délais d'exécution des instructions.

Malgré ses avantages en termes de tolérance aux fautes, les raisons principales pour lesquelles nous n'avons pas exploité le LEON-FT sont les suivantes : d'un coté, il est très sophistiqué pour notre application cible. Nous rappelons que l'application mécatronique cible est dédiée au contrôle et à la commande et ne demande pas énormément de calcul. De l'autre coté, en termes de tolérance aux fautes, le LEON se base principalement sur la redondance et l'utilisation de la technique de TMR (*Triple Modular Redundancy*). En effet, rien ne nous empêche d'utiliser d'autres types de mécanismes de protection. De plus, un des buts futurs du laboratoire est de concevoir des systèmes multi-processeurs tolérants aux fautes embarqués

dans un NOC (*Network On Chip*). Il nous faudra ainsi un processeur de moindre complexité que le LEON pour pouvoir embarquer plus de microprocesseurs dans le NOC.

## D. Conclusions :

Dans ce chapitre, nous avons introduit les notions de sûreté de fonctionnement et de sécurité. La sûreté de fonctionnement d'un système informatique est la propriété qui permet de placer une confiance justifiée dans le service qu'il délivre. Nous avons ensuite donné les concepts de base et terminologie de sûreté de fonctionnement des systèmes. Les attributs, entraves et moyens de sûreté ont été également définis. Ces derniers sont résumés par la figure suivante :

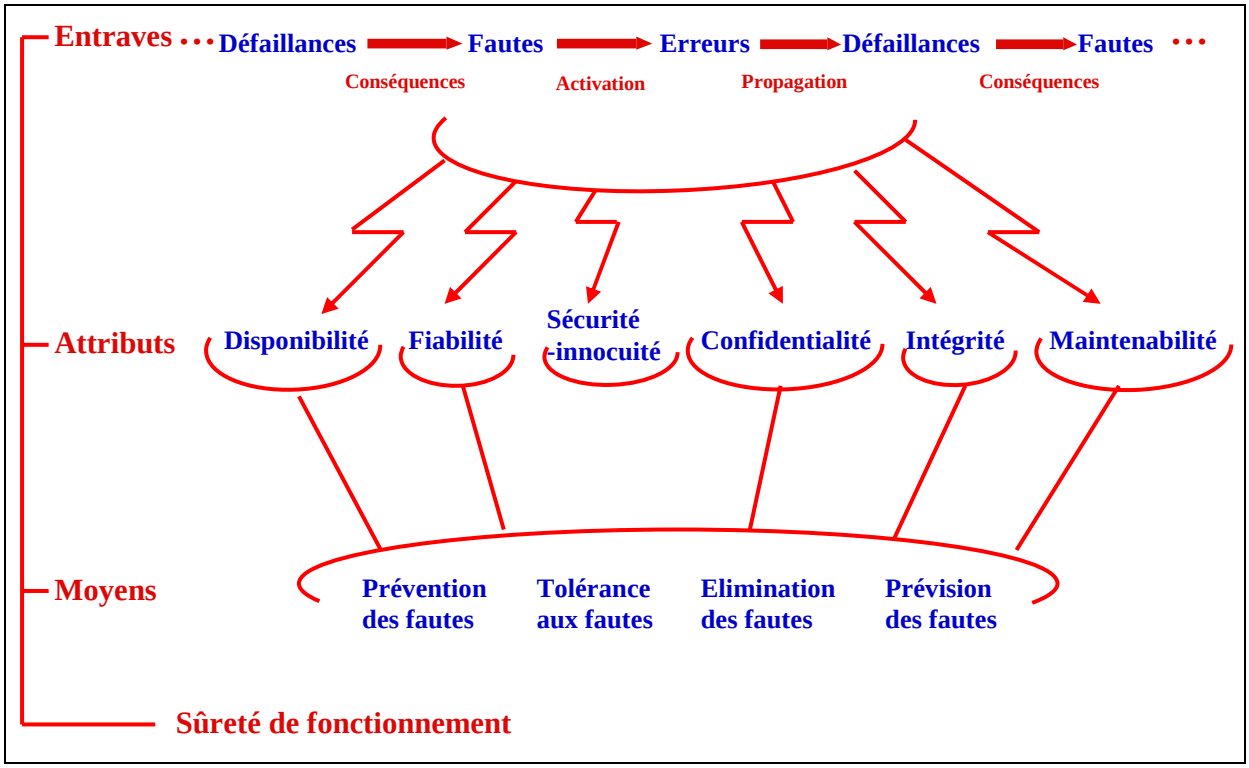


**Figure 1.11 : L'arbre de la sûreté de fonctionnement**

Nous avons donné un ensemble d'erreurs susceptibles d'attaquer les éléments de mémoires, à savoir les effets d'une particule isolée SEE (*Single Event Effect*). Et nous nous sommes intéressés aux fautes transitoires, non permanentes en particulier les perturbations par une particule isolée SEU (*Single Event Upset*) ainsi que ses cibles, ses conséquences et ses mécanismes de protection.

Comme nous le constatons, le premier chapitre nous a permis d'avoir les connaissances nécessaires nous permettant d'aborder la question de sûreté de fonctionnement des

architectures de processeurs plus aisément. Notre objectif étant de proposer une méthodologie de conception d'architectures de processeurs sûres de fonctionnement, nous nous penchons dans le chapitre 2 sur certains aspects parmi lesquels nous citons les solutions architecturales possibles, les besoins en terme d'outils logiciels en vue d'évaluation et d'amélioration de la sûreté et les besoins en terme de techniques de protection.

# Références


— [ACTEL] “ACTEL soft/firm errors glossary”, consulté le 09/03/2009, http://www.actel.com/products/solutions/ser/glossary.html

— [AERO] “Single event effects testing”, http://www.aero.org/capabilities/seet/primer.html consulté le 09/03/2009.

— [AVIZ00] A. Avizienis, J-C. Laprie and B. Randell : “Fundamental Concepts of Dependablity”, in Proc. 3rd IEEE Information Survivability Workshop (ISW-2000), Boston, Massachusetts, USA, October 2000, pp. 7-12.

— [AVIZ04] A. Avizienis, J-C. Laprie, B. Randell and C. Landwehr : “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Trans. on Dependable and Secure Computing, vol. 1, no. 1, Jan-March 2004, pp. 11-33.

— [BERL68] E.R. Berlekamp (Bell Labs): “Algebric coding theory”, Mc Graw-Hill, Series in systems science (1968).

— [CGTN05] Commission générale de terminologie et de néologie - NOR : CTNX0407916K - “Vocabulaire des sciences et techniques spatiales”, JO du 30-01-2005, pp. 1625-1629, texte n° 43.

— [CHEN83] C.L. Chen: “Error-Correcting Codes with Byte Error-Detection Capability”, IEEE Transactions on Computers, Volume C-32, Issue 7, July 1983, pp. 615 – 621.

— [CHEN84] C.L. Chen and M.Y. Hsiao: “Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review”, IBM Journal of Research and Development, Volume 28, Number 2, March 1984, pp. 124 – 134.

— [DILG03] E, Dilger, M, Gulbins, T, Ohnesorge and B, Straube : “On a Redundant Diversified Steering Angle Sensor”, in Proc. 9th IEEE International On-Line Testing Symposium, IOLTS’03, July 2003, pp. 191- 196.

— [DILG04] E. Dilger, R. Karrelmeyer and B. Straube : “Fault tolerant mechatronics”, in Proc. 10th IEEE International On-Line Testing Symposium, IOLTS’04, July 2004, pp. 214- 218.

— [FAUR05] F. Faure : “Injection de Fautes Simulant les Effets de Basculement de Bits Induits par Radiation”, Thèse de Doctorat à l’Institut National Polytechnique de Grenoble - spécialité Microélectronique préparée au laboratoire TIMA, soutenue le 14 novembre 2005.

— [GAIS00] J. Gaisler : “LEON-1 Processor - First Evaluation Results”, European Space Components Conference (ESCCON 2000), 2000.

— [GAIS02] J. Gaisler : “A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture”, in Proc. International Conference on Dependable Systems and Networks (DSN'02), 2002.

— [GAIS03] J. Gaisler : “Preparations for next-generation SPARC processor”, in Proc. Workshop on Spacecraft Data Systems, May 2003.

— [GIRA04] J. Girard, D. Tabet : “Introduction aux notions de sûreté de fonctionnement et de sécurité”, Revue de l’Électricité et de l’Électronique, No. 11, Déc. 2004, pp. 93-94.

— [GIRA07] M. Giraud : “Sûreté de fonctionnement des systèmes – Sûreté informationnelle, l’apport des codes”, Techniques de l’ingénieur, E3 853, Nov. 2007, pp. 1-21.

— [HAMM50] R.W. Hamming: “Error Detecting and Error Correcting Codes”, The Bell System Technical Journal 26, 2 (1950), pp. 147–160.

— [ISER02] R. Isermann, R. Schwarz and S. Stolzl : “Fault-tolerant drive-by-wire systems”, in IEEE Control Systems Magazine, vol. 22, issue 5, October 2002, pp. 64- 81.

— [KATZ01] R. B. KATZ, “Tutorials for Programmable Logic and Military/Aerospace Systems FPGAs in Space Environment and Design Techniques”, presented at the NASA Goddard Space Flight Center June 25, 2001.

— [KOCH04] H.-D. Kochs : “Key Factors Key dependability of mechatronic units”, in 28th IEEE Annual International Computer Software and Application Conference (COMPSAC’04), September 2004, Hong Kong.

— [LAPR95] J-C. Laprie : “Dependability of computer systems : concepts, limits, improvements”, in Proc. 6th International Symposium on Software Reliability Engineering, October 1995.

— [LAPR04] J-C. Laprie : “Sûreté de fonctionnement des systèmes : concepts de base et terminologie”, Revue de l’Électricité et de l’Électronique, No. 11, Déc. 2004, pp. 95-105.

— [McEL77] R.J. Mc Eliece (Jet Propulsion Lab., Caltech): “The Theory of Information and coding”, Addison-Wesley (1977).

— [MEIX08] A. Meixner, M.E. Bauer and D.J. Sorin : “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores”, IEEE Micro, vol.28, issue 1, February 2008, pp. 52–59.

— [MERA07] N. Merabtine, D. Sadaoui et M. Benslama : “Contribution à l’étude du phénomène de latchup induit dans les circuits integrés embarqués dans un environnement radiatif spatial ”, Romanian Journal of Physics, vol. 52, no. 1-2, 2007, pp. 119-129.

— [PIES92] S. Piestrak : “Design of self testing checkers for unidirectional errors detecting codes”, Monographie univ. Techn., Varsovie 1992.

— [RIEM01] R. A. Riemenschneider and Steve Dawson : “Dependability Co-Design”, in Proc. Workshop on New Visions for Software Design and Productivity: Research and Applications, December 13-14, 2001, Nashville, Tennessee, USA.

— [ROUC92] G. Rouchouse (1992) : “Sûreté des automatismes”, Publications CETIM, Mécanique et Productique, ISBN 2-85400-205-5, 139p.

— [SADA05] D.Sadaoui, A.Benslama et M.Benslama : “Étude de l’aléa logique (SEU) induit dans les mémoires SRAM”, in Proc. 3rd International

Conference: Sciences of Electronic Technologies of Information and Telecommunications (SETIT 2005), MARCH 2005 – TUNISIA.

— [SHYA06] S. Shyam et al. “Ultra Low-Cost Defect Protection for Microprocessor Pipelines”, ACM SIGPLAN Notices, vol.41, issue 11, November 2006, pp. 73-82.

— [SING06] K. Singh, A. Agbaria, D-I. Kang and M. French : “Tolerating SEU Faults in the Raw Architecture”, in Proc. of the 3rd International Workshop on Dependable Embedded Systems, Austria, November 2006, pp. 35-40.

— [SLEG99] T. J. Slegel et al.: “IBM’s S/390 G5 Microprocessor Design”, IEEE Micro, vol.19, issue 2, March/April 1999, p. 12–23

— [STOJ01] M.K. Stojcev, G.Lj. Djordjevic and M.D. Krstic : “A hardware mid-value select voter architecture”, ELSEVIER Microelectronics Journal 32 (2001), pp. 149–162

— [THOU07] D. Thouvenot: “Vulnérabilité des équipements électroniques aux effets singuliers des particules”, Conférence Salon Hyper & RF, Paris, Mars 2007.

— [ZARA07] H.R. Zarandi et al. : “Fast SEU Detection and Correction in LUT Configuration Bits of SRAM-based FPGAs”, Parallel and Distributed Processing Symposium 2007, IPDPS 2007. March 2007, pp. 1 – 6.

# Chapitre 2

# Étude qualitative : élaboration de la méthodologie

Le développement des technologies microélectroniques entraîne une utilisation de plus en plus importante des circuits intégrés dans les systèmes mécatroniques. La sûreté de fonctionnement offerte par ces circuits est un paramètre très intéressant à étudier dans la mesure où plein d'applications mécatroniques nous entourent comme nous l'avons cité dans l'introduction de ce mémoire. Etant donné que l'élément de base de ces circuits est bien entendu le processeur, le choix de ce dernier, son type d'architecture, son jeu d'instruction…etc., doivent être bien étudié en tenant compte de l'aspect de la sûreté de fonctionnement qui nous est prioritaire. C'est pour cela que dans le premier chapitre, nous avons présenté les notions nécessaires nous permettant d'aborder la question de sûreté de fonctionnement des architectures de processeurs beaucoup plus aisément. Ainsi, nous introduisons dans ce second chapitre notre méthodologie principalement basée sur le développement d'un émulateur de processeur. Nous montrons son utilité dans un objectif de sûreté de fonctionnement. Évidemment, l'émulateur ne sera se développer sans définir et justifier le type d'architecture ainsi que sa structure interne et son jeu d'instructions, ce qui fera aussi l'objet de ce chapitre. Enfin, nous détaillons la technique de protection proposée en tant que moyen pour la sûreté. Nous commençons alors par introduire la méthodologie.

# A. Introduction de la méthodologie

Notre objectif est de proposer et de valider une méthodologie de conception d'architectures de processeur sûres de fonctionnement. Afin d'y parvenir, la méthode que nous avons adoptée consiste à étudier le comportement fonctionnel du processeur avant d'entrer dans les détails de sa structure interne telle qu'elle sera implantée dans un FPGA par exemple. Cette étude fonctionnelle, qui précède une étude architecturale, est réalisée grâce au développement d'un émulateur du processeur. En effet, cet émulateur décrit l'évolution des états internes du processeur et calcule le nombre de cycles nécessaires pour l'exécution d'un programme. De plus, si nous connaissons le nombre de cycles d'horloge de chaque instruction, nous pouvons déduire la durée nécessaire en nombre de cycles d'horloge pour l'exécution d'un programme. L'émulateur permet d'évaluer les états internes et les durées d'exécution.

Toutefois, afin de réaliser cette étude fonctionnelle, nous avons besoin, à part l'émulateur, de *benchmarks* représentatifs des programmes embarqués dans le processeur. Ces *benchmarks* seront ainsi testés dans l'émulateur. D'une part, ces tests nous permettront d'obtenir des statistiques, notamment sur les accès mémoire, les instructions les plus fréquentes, le nombre de cycles en fonction des différentes stratégies concernant le nombre de bus (deux ou trois bus, nous en discuterons ultérieurement). Ces statistiques nous serviront alors à affiner les choix architecturaux. D'autres part, certaines erreurs matérielles ont des conséquences fonctionnelles et peuvent donc être analysées avec l'émulateur. Elles ont, entre autres, pour conséquence une altération du déroulement du programme et une modification des états du processeur. Pour y remédier, des techniques de protection pourront ainsi être ajoutées dans les *benchmarks*. Ceci nous sera d'une grande utilité dans la mesure où notre priorité est la sûreté de fonctionnement.

La méthodologie de conception proposée et illustrée dans la Figure 2.1, consiste à développer un émulateur de processeur intégrant les résultats et contraintes obtenus suite à l'étude de différents critères liés à la conception de l'architecture. Nous pouvons citer parmi ces critères :

- Le contexte mécatronique du projet et ses implications sur l'environnement,
- La sûreté de fonctionnement des architectures (modélisation des erreurs et de techniques de protection)

— Le protocole d'échange entre le processeur et l'ensemble des capteurs, actionneurs et utilisateur
— Le faible coût
— La contrainte de durée du projet
— Le besoin en jeu d'instructions.

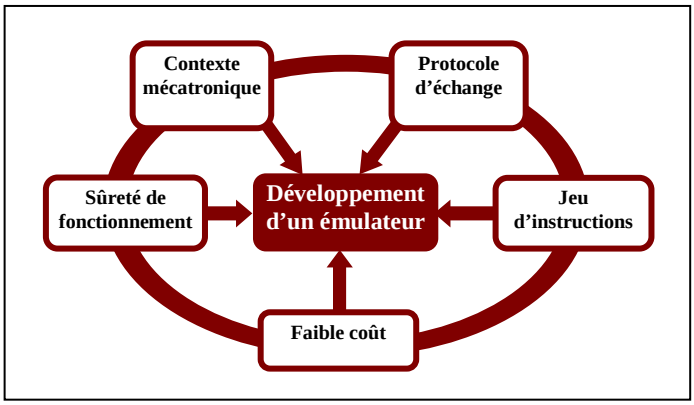


**Figure 2.1 : Méthodologie de conception**

Un intérêt particulier est consacré aux deux point suivants : l'aspect sûreté de fonctionnement et le contexte mécatronique. Ceci à travers une identification des défaillances affectant les circuits dans un environnement mécatronique ainsi que les techniques de protection. De ce fait, plusieurs facteurs de sûreté sont pris en compte [KOCH04], en particulier pour les applications mécatroniques [ISER02], [DILG03], [DILG04].

Le développement de l'émulateur doit être précédé par le choix du type d'architecture. Parmi les architectures disponibles, nous citons les architectures classiques CISC ou RISC ainsi que l'architecture MISC (*Minimal Instruction Set Computer*). Pour cela, une étude préalable et comparative de ces trois types architectures s'avère nécessaire pour pouvoir justifier la démarche suivie pour le choix de l'architecture que nous allons adopter.

# B. Démarche suivie pour définir les choix architecturaux

## B.1. Étude comparative des architectures CISC, RISC et MISC

Avant de concevoir notre processeur, une première étude consiste à choisir le type d'architecture. Les deux principales architectures répandues sont le CISC (*Complex Instruction Set Computer*) et le RISC (*Reduced Instruction Set Computer*). Dans l'architecture CISC, nous distinguons différents modes d'adressage et un nombre important d'instructions. Ces dernières sont de longueur variable. Pour décrire un algorithme, nous avons besoin de peu

d’instructions. Cependant, il y a une certaine difficulté pour réaliser un compilateur et la surface est assez importante. Quant à l’architecture RISC, les modes d’adressage sont limités, nous disposons de relativement peu d’instructions qui ont une longueur figée. Pour cela, un algorithme écrit pour un processeur RISC exige plus d’instructions que le CISC et par conséquent la programmation en assembleur est plus difficile. À ces deux architectures CISC et RISC s’ajoute une architecture originale de type MISC (*Minimal Instruction Set Computer*). Celle-ci présente la particularité de posséder très peu d’instructions et d’exiger peu de ressources matérielles. Donc, dans le cadre de la tolérance aux fautes, une duplication ne sera pas coûteuse. Cette architecture de processeur est généraliste et réduite à l’essentiel comme c’est le cas du processeur à pile où la pile remplace les registres. Par rapport à l’architecture RISC, le jeu d’instructions d’un MISC est minimisé, résultant en un processeur moins coûteux avec des performances raisonnables, comme dans le cas du processeur M17 [KOOP89]. En effet, le M17 a été conçu par le *Minimum Instruction Set Computer, Inc.* [KOOP89], comme étant un microprocesseur à faible coût. Afin d’atteindre cet objectif de faible coût, le M17 garde ses deux piles dans la mémoire programme avec en plus certains autres éléments de la pile dans le cœur du processeur. D’autres compromis et améliorations au niveau de la conception ont été faits afin de garder un coût faible tant pour la production du circuit intégré que pour le circuit lui-même tout en maintenant des hautes performances du système.

Afin de comparer ces différentes architectures, nous nous basons sur certaines études comparatives déjà faites. *L’Association of Computing Machinery* a réalisé une des comparaisons la mieux détaillée basée sur les structures de contrôle, les structures arithmétiques et logiques, les structures de mémoire, les entrées/sorties, la communication de données et les circuits intégrés utilisés [SHAO03].

Dans la mesure où les architectures CISC et RISC sont connues, nous allons plutôt détailler l’intérêt du MISC par rapport au RISC. En effet, l'augmentation de la vitesse dans le processeur RISC crée un gros ralentissement entre le processeur et la mémoire qui est plus lente. Pour augmenter la vitesse de l'accès à la mémoire, il est nécessaire d'utiliser de la mémoire cache pour les buffers d’instructions et les flux de données. Les processeurs RISC sont relativement inefficaces dans le traitement des appels et retours des sous-routines. Ce traitement est un critère critique pour l’évaluation de la capacité du processeur à supporter des langages de haut niveau. Certains processeurs RISC utilisent un registre supplémentaire pour les appels et retours des sous-routines. Toutefois, ce registre doit être suffisamment large pour

qu'il puisse contenir certains paramètres (entrées, sorties, variables locales). Ce registre exploite la ressource la plus précieuse dans le processeur RISC et ralentit par conséquent le système durant le changement de contexte. Le principe de simplicité n'est donc pas suffisamment retrouvé pour en tirer un maximum de profit. Ainsi, les architectures MISC explorent la simplicité au maximum par la prise en charge d'une trentaine d'instructions généralement. Comme exemple, nous pouvons citer le MuP21 comportant quatre groupes d'instructions [MOOR95], [TING95] : des instructions de transfert de données, des instructions de manipulation de mémoire, des instructions arithmétiques et des instructions de manipulation de registre. Jusque là, seulement 24 instructions sont implantées dans ce processeur MISC, laissant de la place pour un éventuel ajout de nouvelles instructions. Par exemple, l'opération soustraction peut ainsi être synthétisée par le complément et l'addition. L'opération OU peut aussi être synthétisée par le complément, l'opération ET et l'opération XOR. Les applications potentielles pour le MuP21 incluent les jeux vidéo, les contrôleurs de disques durs, les contrôleurs de robots, les ordinateurs de poche …etc. Ainsi, nous pouvons résumer la différence entre ces trois architectures dans le tableau suivant [SHAO03] :

| | **CISC** | **RISC** | **MISC** |
|---|---|---|---|
| **Complexité de l'instruction** | | | |
| Complexe | x | | |
| Moyenne | | x | |
| Simple | | | x |
| **Type de l'instruction** | | | |
| Avancée | x | | |
| Entre les deux | | x | |
| Basique | | | x |
| **Taille de l'instruction (1)** | | | |
| Large | x | | |
| Moyenne | | x | |
| Petite | | | x |
| **Taille de l'instruction (2)** | | | |
| Uniforme | | x | x |
| Non-uniforme | x | | |
| **Flux de contrôle (*flow control*)** | | | |
| Asynchrone | | | |
| Synchrone | x | x | x |
| **Flux de donnée (*data control*)** | | | |
| Série | x | | x |
| Parallèle | | | |
| Série/parallèle | | x | |
| **Performance dominante (*Performance Domination*)** | | | |
| Matérielle | x | | |
| Logicielle | | x | x |
| Matérielle/logicielle | | | |

**Tableau 2.1 : Comparaison des architectures CISC, RISC et MISC**

Dans l'architecture CISC, nous distinguons différents modes d'adressage et un nombre important d'instructions. Ce qui fait que la surface devient importante puisque le contrôleur requiert plus de matériel. Le nombre d'instructions diminue chez les RISC et est minimal dans les architectures MISC. Ces instructions sont de longueur variable dans les architectures CISC alors qu'ils sont de longueur fixe dans les deux autres architectures.
Après cette étude comparative nous nous proposons de définir et expliquer nos choix en terme type d'architecture, ce qui fera l'objet du paragraphe suivant.

## B.2. Convergence des choix - Vers une architecture à piles

Suite à la comparaison faite entre les 3 familles d'architectures (CISC, RISC et MISC) [SHAO03], et sachant que le paradigme de sûreté guidant notre recherche est la simplicité, comme l'indique les figures ci-dessous (Figure 2.2 et Figure 2.3), notre choix s'est orienté vers l'architecture MISC du fait de ses avantages, notamment en termes de surface, de nombre minimal d'instructions et de longueur fixe des instructions. Ainsi, en tenant particulièrement compte de la sûreté de fonctionnement, nous avons choisi une architecture à pile pour sa simplicité et sa flexibilité. En effet, la simplicité architecturale permet de réduire la durée de conception et surtout de rendre plus facile la protection du processeur. D'une part, sa surface est plus faible par rapport aux surfaces des autres architectures. Sachant que nous trouvons plus de ressources combinatoires que de ressources de mémorisation chez les MISCs et que les SEUs interviennent beaucoup plus dans les points mémoires [SPRI01], la probabilité que les MISCs soient exposée aux erreurs SEUs est plus faible que chez les autres architectures. D'autre part, dans un contexte de sûreté de fonctionnement et remarquant la faible quantité d'informations mémorisées dans un processeur MISC, nous avons déjà une idée sur un moyen de recouvrement qui sera idéalement exploité dans ce cas, à savoir la reprise. Comme nous l'avions présenté dans le chapitre 1, la reprise est une technique générale qui consiste en un retour en arrière vers un état antérieur dont on sait qu'il est correct. Elle nécessite donc la sauvegarde de l'état. Elle consiste à choisir des points de reprise, pendant les quels nous faisons un sauvegarde périodique de l'état du système en utilisant des mécanismes de mémorisation qui protègent les informations, vis à vis des effets des erreurs tolérées et en tenant compte des problèmes de cohérence de l'état global sauvegardé. Ainsi, nous déterminons l'état global cohérent le plus proche et nous reprenons les traitements. Le fait que le MISC a une faible quantité de mémorisation alourdira moins les délais de reprise par rapport aux autres architectures de type RISC et MISC. Ces derniers ont une plus grande quantité de mémorisation. De ce fait, la restauration d'un état correct après

l'occurrence d'une erreur est plus simple chez les MISCs que chez les RISCs et CISCs. Donc, plusieurs techniques de protection, qu'elles soient basées sur la duplication ou la sécurisation des différents éléments constitutifs du cœur du processeur, des bus et des mémoires deviennent envisageables. Nous pouvons ainsi résumer la différence entre les trois architectures en y ajoutant les critères de sûreté de fonctionnement (SdF) :

| | | CISC | RISC | MISC |
|---|---|---|---|---|
| **Critères de SdF** | **Surface** | ☹ | 😐 | ☺ |
| | **Nombre d'instruction** | ☹ | 😐 | ☺ |
| | **Longueur d'instruction** | Variable | Figée | Figée |
| | **Duplication** | ☹ | 😐 | ☺ |
| | **Protection du cœur** | ☹ | 😐 | ☺ |

**Tableau 2.2 : Comparaison brève entre CISC, RISC et MISC en prenant en compte l'aspect de sûreté de fonctionnement**

Le principe de choix d'un processeur se basant sur la simplicité est schématisé à travers les figures Figure 2.2 et Figure 2.3.

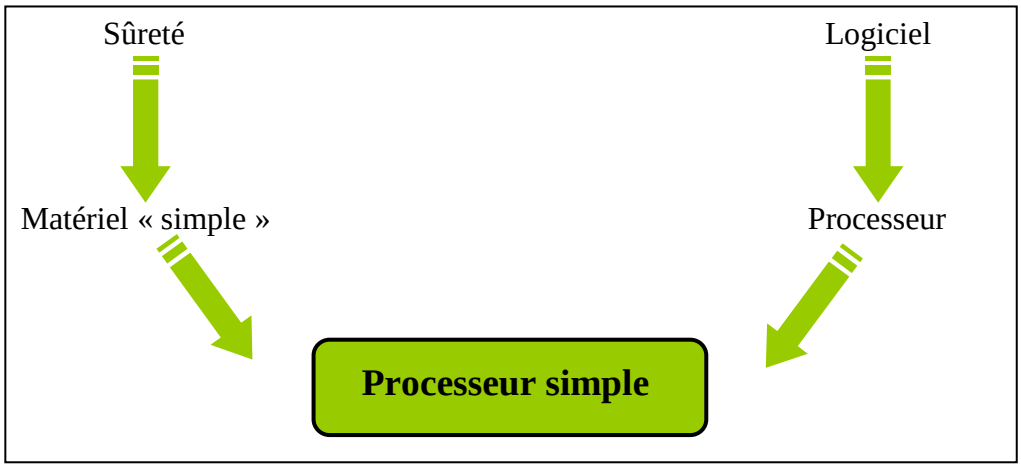


**Figure 2.2 : Principe de choix du processeur se basant sur la simplicité**

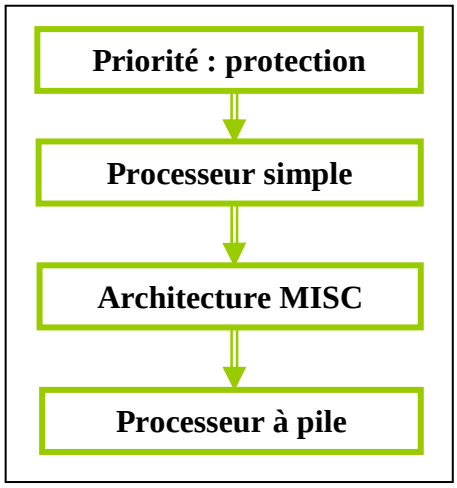


**Figure 2.3 : Démarche de choix du processeur à pile**

D'un coté, pour sécuriser un matériel en se basant sur la duplication, nous avons intérêt à ce que ce matériel soit le plus simple possible. De l'autre coté, pour commander une application, nous aurons besoin d'un logiciel embarqué dans un processeur. La fusion de ces deux conditions requiert alors à ce que ce processeur soit le plus simple possible (Figure 2.2). Concernant les critères de sûreté de fonctionnement, comme un des principaux recours pour la protection du cœur du processeur se base sur la duplication, nous pouvons dire qu'il est tout à fait clair que, plus le matériel est simple, plus sa duplication devient facile, moins coûteuse et moins compliquée. De ce fait, l'architecture MISC devient la plus appropriée, en particulier l'architecture à piles (Figure 2.3).

Nous avons décrit les raisons de choix de l'architecture MISC et du processeur à pile. Dans la mesure où les applications cibles sont des applications de contrôle et de commande et ne demandent pas de calcul poussé, il n'est pas nécessaire d'avoir un processeur sophistiqué. En plus, étant donné que la stratégie de protection consiste à faire une sauvegarde et une restauration des éléments constituant le cœur du processeur, il est plus judicieux de concevoir une architecture minimaliste.

Dans la suite, nous nous proposons de présenter les particularités internes de l'architecture à piles. Nous donnons tout d'abord les différences en termes de nombre de piles internes, de la profondeur de la pile et de nombre d'opérandes. Ensuite nous exposons le modèle canonique de la machine à pile et ses différents constituants. Nous nous basons sur ce modèle pour schématiser ultérieurement la structure interne de notre propre processeur.

## B.3. Structure interne des architectures à piles - modèle canonique d'une architecture à deux piles

La différence dans la structure interne d'une architecture à pile est résumée dans la figure 2.4, définissant ainsi l'espace de conception d'un processeur à piles. Concernant le nombre d'opérandes, nous trouvons des architectures à zéro, un ou deux opérandes. La pile peut être petite ou profonde. Enfin, nous pouvons avoir un processeur à une seule pile, à une seule pile avec des registres auxiliaires, ou à deux piles, etc.
Étalons alors ces différences et détaillons leurs spécificités les unes des autres.

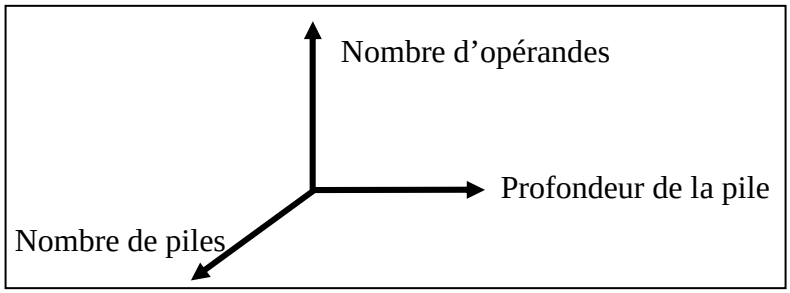


**Figure 2.4 : Espace de design d'un processeur à pile**

Pour les architectures à 0-opérande, les instructions sont soit spécifiques aux sommets de la pile, soit spécifiques au sommet et au sous sommet. Le code instruction est utilisé uniquement pour l'instruction. Huit bits nous permettent ainsi d'obtenir 256 instructions. Le décodage de l'instruction est simplifié puisqu'il n y a pas de mode d'adressage. Pour l'accès à une donnée au fond de la pile, nous aurons la possibilité soit de faire des copies vers la mémoire, soit d'avoir un adressage matérielle de la pile ou soit d'utiliser une seconde pile.

Pour les architectures à 1-opérande, le sommet de la pile est utilisé comme second opérande. La recherche de l'opérande se fait en parallèle avec l'opération effectuée sur la pile. Ces architectures supportent le mode d'adressage 0-opérande.

Enfin, pour les architectures à 2-opérandes, nous avons un opérande source et un opérande destination. Nous avons plus de flexibilité puisque nous avons deux opérandes. Cependant, l'architecture matérielle est plus compliquée.

Maintenant que nous avons vu les particularités selon le nombre d'opérandes, intéressons nous aux avantages, inconvénients et utilités de la présence d'une pile unique ou de piles multiples dans le processeur, ce qui est résumé dans le Tableau 2.3 suivant.

| | Utilité | Avantages | Inconvénients |
|---|---|---|---|
| **Pile unique** | - Adresses de retour des fonctions<br>- Paramètres de fonctions<br>- Sauver le contexte d'une fonction lors d'un appel ou d'une interruption | - Simplicité de point de vue matériel<br>- Facilité de gestion (gestion d'un seul bloc mémoire) | - Combiner dans une seule pile les paramètres de fonction et leurs adresses de retour (pas pratique pour la gestion) |
| **Piles multiples** | - Une pile pour les adresses de retour<br>- Une deuxième pile pour l'évaluation des expressions et/ou passage des paramètres de fonctions | - Séparation des adresses et des données<br>- Éviter certaines copies obligatoires de données<br>- Vitesse : accès à des valeurs en un seul cycle (appel et retour des programmes en parallèle avec les opérations sur les données) | - Deux piles à gérer |

**Tableau 2.3 : Comparaison entre un processeur à une seule pile et processeur à piles multiples**

Nous venons d'exposer l'espace de conception d'une architecture à pile. La structure interne de cette architecture est spécifique selon le nombre de piles, la profondeur de la pile et le nombre d'opérandes. Intéressons-nous au modèle canonique d'une architecture à deux piles qui est donné par la Figure 2.5 [KOOP89].

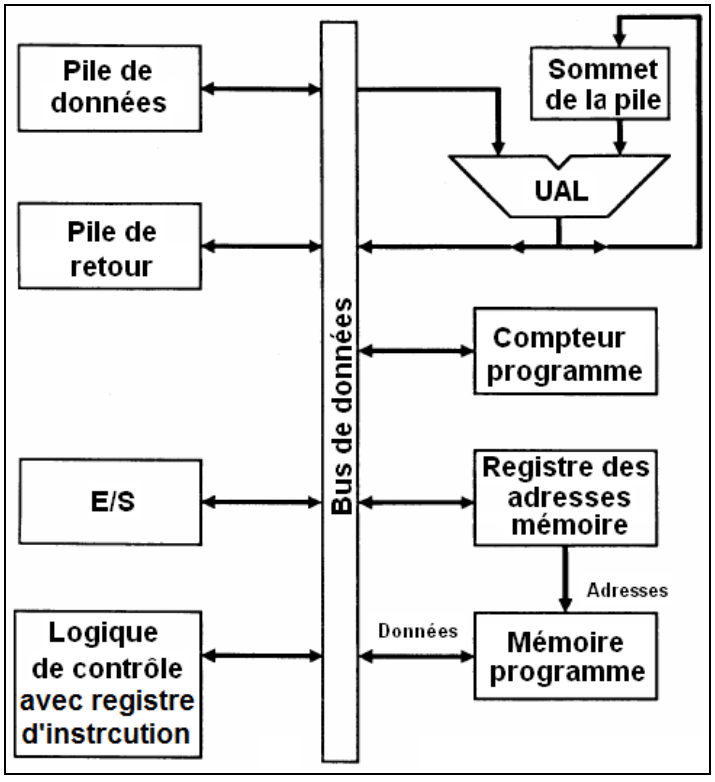


**Figure 2.5 : Schéma d'une machine canonique à deux piles**

Chaque bloc représente une ressource logique correspondante ainsi aux composants minima nécessaires pour la conception d'un processeur 0-opérande à piles multiples. Ces composants sont : le bus de données, la pile de données, la pile de retour, l'unité arithmétique

et logique avec son registre le sommet de la pile, le compteur programme, la mémoire programme avec son registre des adresses mémoire, la logique de contrôle avec le registre d'instruction, et enfin un bloc entrées/sorties.

Pour des raisons de simplicité, la machine canonique a un seul bus connectant toutes les ressources. D'autres processeurs peuvent avoir plus qu'un seul bus de données pour permettre de faire les opérations d'aller chercher les données et d'exécuter l'instruction en parallèle (FETCH et EXECUTE en parallèle).

La pile de données est une mémoire avec un mécanisme interne gérant une pile LIFO (*Last In First Out*) dernier entré, premier sorti. Une implémentation commune de ceci peut être une mémoire conventionnelle avec un compteur/décompteur utilisé pour la génération d'adresse.

La pile de données permet deux opérations : empiler (PUSH) et dépiler (POP). L'instruction PUSH alloue une nouvelle zone mémoire au sommet de la pile et y écrit la valeur se trouvant sur le bus de données. L'instruction POP place la valeur du sommet de la pile sur le bus de données et supprime la zone mémoire dans laquelle elle était, rendant la zone mémoire qui la suit, le nouveau sommet de la pile qui servira à la prochaine opération.
La pile de retour est implémentée de la même manière que la pile de donnés. La seule différence est que celle-ci est utilisée pour sauvegarder les adresses de retour des fonctions au lieu des opérandes.

L'unité arithmétique et logique (UAL) effectue les opérations arithmétiques et logiques sur une ou deux données. Dans le cas de deux données, la première donnée se trouve dans le registre sommet de la pile (schématisée directement au dessus de l'UAL de la Figure 2.5.). Ainsi, l'élément au sommet du bloc pile de données est alors le second élément (sous-sommet) de la pile de données. Ça permettrait d'accéder une seule fois au bloc pile de données dans le cas d'opérations tel que l'addition, une opération qui bien entendu utilise deux éléments : le sommet et le sous-sommet de la pile.

Le compteur programme (CP) charge l'adresse de la prochaine instruction à exécuter. Il peut être chargé à partir du bus dans le cas de branchement comme il peut être incrémenté pour aller chercher une suite séquentielle d'instructions à partir de la mémoire programme.
Pour accéder à la mémoire programme, l'adresse qu'on veut lire ou écrire est tout d'abord écrite dans le registre des adresses mémoire. Ensuite, dans le prochain cycle, la mémoire programme est lue ou écrite sur le bus de données.

Comme c'est le cas de nombreuses conceptions, l'issue des entrées/sorties est relative à chaque modèle. Nous nous contentons de dire qu'un bloc d'entrées/sorties existe dans le modèle canonique [KOOP89].

Maintenant que les composants de base de la machine canonique à deux piles sont définis, et que les différences de la structure interne sont précisées, nous présentons la structure interne de l'architecture à pile que nous adaptons dans la suite de la thèse, tout en détaillant et expliquant les raisons de ces choix.

## B.4. Description de la structure interne de l'architecture MISC adoptée - Vers une architecture à deux piles

L'architecture à pile étant choisie et justifiée, nous avons donné une synthèse bibliographique sur les architectures à pile et une énumération des différences en terme de spécificités internes. Cette synthèse nous permet d'aborder la question de la structure interne de l'architecture à pile beaucoup plus aisément. Ainsi, nous exposons les détails de la structure interne du processeur à piles et les raisons de ces choix. Ce processeur doit répondre à nos besoins. Nous rappelons que nous avons besoin d'un processeur sûr et simple, dont la duplication est non coûteuse et la protection du cœur plus facile :

- Processeur à deux piles, l'une dédiée aux données appelée, pile de données DS '*Data Stack*' et la deuxième dédiée aux adresses de retour appelée, pile de retour RS '*Return Stack*' utilisée lors d'appels de fonctions, d'interruptions ou lors de copies temporaires.
- Ces piles sont gérées en mémoire externe au processeur '*Data Stack Memory*' et '*Return Stack Memory*' pour ne pas avoir de restriction sur leurs profondeurs. Ces piles sont adressées par deux pointeurs internes au cœur : pointeur pile de données DSP '*Data Stack Pointer*' et pointeur pile de retour RSP '*Return Stack Pointer*'.
- Le sommet et le sous-sommet de la pile de données TOS '*Top-Of-Stack*' et NOS '*Next-Of-Stack*' ainsi que le sommet de la pile de retour TORS '*Top-Of-Return-Stack*' sont internes au processeur.
- Instructions 0-opérande puisque la très grande majorité des instructions manipule directement les sommets des piles.
- Instructions de longueur 8 bits étant donné qu'elles comportent très rarement un opérande et que notre besoin en instructions est très inférieur à 256 instructions.
- Bus de données de largeur 16 bits.

Cette démarche que nous venons de décrire est résumée dans la Figure 2.6 suivante :

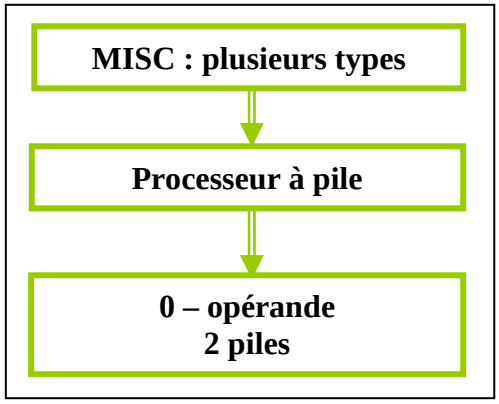


**Figure 2.6 : Démarche de choix du processeur à pile**

Dès lors, l'architecture étant choisie, la structure interne également, nous nous proposons de concevoir le jeu d'instructions conformément aux besoins d'une application mécatronique dédiée au contrôle et à la commande. À noter que nous avons ajouté d'autres instructions permettant de sauvegarder les éléments constituant le cœur du processeur. Ces instructions servent au bon déroulement de l'opération de la poursuite (recouvrement arrière) en tant que moyen de protection.

## B.5. Conception du jeu d'instructions

Le Tableau 2.4 énumère les instructions et les regroupe par type.

| **Type d'instructions** | **Détails** |
|---|---|
| Instructions de pile de données | DROP, LIT a (octet), DLIT a (mot), DUP, SWAP, OVER, ROT |
| Instructions arithmétiques | ADD, ADC (avec reste), SUB, SUBC (avec reste), NEG, MUL, DIV, MOD, INC, DEC, SIGNE (TOS devient $0000 ou $FFFF selon son signe précédent) |
| Instructions logiques | AND, OR, XOR, NOT |
| Instructions entre les 2 piles | R2D: couper/coller entre TORS et TOS<br>D2R: couper/coller entre TOS et TORS<br>CPR2D: copier/coller entre TORS et TOS |
| Instructions de stockage | FETCH: TOS ← Mémoire explicite [TOS]<br>STORE: Mémoire explicite [TOS] ← NOS |
| Instructions d'adressage | CALL a (a est une adresse (mot)), RETURN, SBRA d (d est un déplacement (octet)), LBRA a, ZBRA d (faire le saut si TOS est nul) |
| Instructions de stockage de pointeurs de pile | Push_DSP (sauvegarder DSP dans le TOS), Push_RSP,<br>Pop_DSP (restaurer DSP du TOS), Pop_RSP |
| Autres instructions | NOP (no operation), HALT |

**Tableau 2.4 : Jeu d'instructions regroupées par type**

Le jeu d'instruction contient les 38 instructions détaillées dans le Tableau 2.4. À partir de ces instructions, le chemin de données est établi. À part les mémoires pile de données et pile de retour introduites ci-dessus, nous utilisons une autre zone mémoire appelée mémoire explicite. Elle est accessible grâce aux instructions STORE/FETCH pour le stockage des données. TOS et NOS servent respectivement comme adresse et donnée. En résumé, nous avons quatre zones mémoires différentes (mémoire programme, mémoire pile de données, mémoire pile de retour et mémoire explicite). Ainsi, l'émulateur est développé à partir du jeu d'instructions initialement conçu, permettant de répondre aux besoins d'une application mécatronique dédiée au contrôle et à la commande. Ce jeu d'instruction a évolué au fur et à mesure de notre avancement dans le développement de l'émulateur et de *benchmarks*. Notons que, pour le développement de l'émulateur, nous nous sommes basés sur les conséquences de chaque instruction sur les états internes du processeur ainsi que sur les zones mémoire. Celui-ci est développé en langage C.

Dans la mesure où les applications cibles sont des applications de contrôle et de commande et ne demandent pas de calcul poussé, il n'est pas nécessaire d'avoir un processeur sophistiqué. En plus, étant donné que la stratégie de protection consiste à faire une sauvegarde et une restauration des éléments constituant le cœur du processeur, il est plus judicieux de concevoir une architecture minimaliste.

Nous avons décrit les raisons de choix du processeur à pile. Nous avons détaillé sa structure interne. Nous venons d'exposer le jeu d'instructions de l'architecture à pile choisie. Il nous reste à définir l'adressage mémoire. En effet, nous rappelons que les piles sont gérées en mémoire externe au processeur : mémoire pile de données et mémoire pile de retour pour ne pas avoir de restriction sur leurs profondeurs. Ces piles sont adressées par les deux pointeurs internes au cœur du processeur : pointeur pile de donnée DSP et pointeur pile de retour RSP. À part ces deux zones mémoire, nous avons défini ci-dessus la mémoire explicite accessible grâce aux instructions STORE/FETCH pour le stockage des données. Ajoutant à ça la mémoire programme, nous obtenons ainsi quatre zones mémoire la mémoire programme, la mémoire de pile de données, la mémoire pile de retour et la mémoire explicite. Divers stratégies sont possibles pour les adresser, ce qui fera l'objet de la partie suivante.

## B.6. Différentes stratégies d'adressage mémoire

Nous nous proposons dans cette partie de donner la structure de la mémoire cache et de son organisation. Celle-ci peut être adressée à travers deux bus comme elle peut être adressée

à travers trois bus. En effet, les instructions qui sont dans la mémoire programme (M1), peuvent manipuler trois autres zones mémoires, la mémoire de pile de données (M2), contenant les éléments de la pile, la mémoire pile de retour (M3), contenant les adresses de retour après l'appel des fonctions et la mémoire explicite (M4) contenant les données mémoire adressées directement (M1, M2, M3 et M4 pouvant se recouvrir partiellement). Deux stratégies sont possibles comme nous le constatons sur les deux figures suivantes.

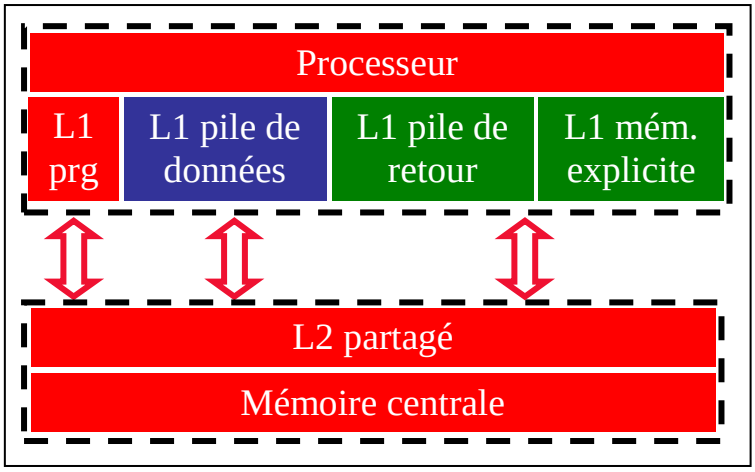


**Figure 2.7 : Stratégie trois bus**

La stratégie à trois bus, illustrée dans la Figure 2.7, consiste alors à adresser M1 seule, M2 seule et M3 et M4 ensemble « dans un seul bloc » puisque nous avons constaté qu'il n'y a aucune instruction qui manipule M3 et M4 en même temps. Ceci nous permet d'avoir une séparation des différents accès ce qui nous évite le risque de conflit entre ces différents accès puisque chaque instruction s'exécutera en un seul cycle.

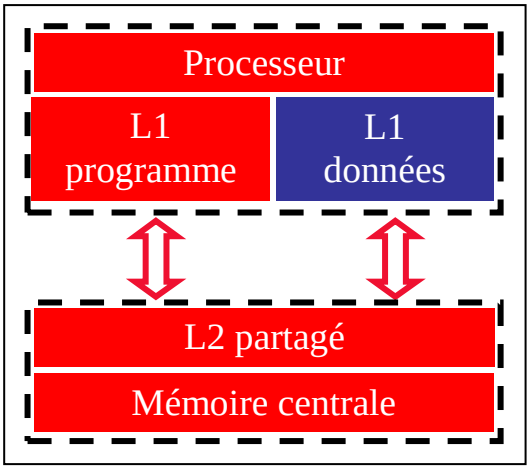


**Figure 2.8 : Stratégie deux bus**

Une autre alternative consiste à unifier deux ou trois mémoires de manière à avoir uniquement deux blocs mémoire et donc deux bus (Figure 2.8), c'est-à-dire unifier M1 et M3, unifier M1 et M4 ou enfin laisser M1 seule et unifier le reste. Cette stratégie a pour conséquence d'avoir un surcoût matériel (des multiplexeurs) servant à la sélection des mémoires. Ajoutant à cela, certaines instructions se voient ralentir dans la mesure où elles traitent des données provenant du même bloc mémoire. Celles-ci, s'exécutant en un seul cycle avec la stratégie trois bus, s'exécutent en deux cycles en stratégie deux bus. Vérifions alors si ce ralentissement a une influence sur la sûreté de fonctionnement. En effet, supposons que ce ralentissement a un surcoût de 50%, que le processeur est jugé sûr pour un taux d'erreurs de une erreur toutes les 70 ms, et que le programme qui y tourne dure 50 ms en stratégie trois bus. Suite à ce ralentissement, ce même programme tourne en 75ms (50% de plus) en stratégie deux bus. Un environnement à une erreur toutes les 70 ms aura pour conséquence la non sûreté du processeur avec une stratégie en deux bus. Dans ce cas, la stratégie trois bus est meilleure. De plus, à part cette perte en performances temporelles à cause de la stratégie deux bus, nous risquons d'avoir d'autres pertes en utilisant éventuellement des méthodes de protection logicielles, ce que nous montrerons dans le chapitre 3. En fait, ces techniques logicielles peuvent ralentir le programme puisque certaines routines de protection seront intégrées dans le *benchmark*. La comparaison de ces performances temporelles et la quantification des surcoûts temporels qu'ils soient dû à la stratégie deux bus ou qu'ils soient dû aux routines de protection feront partie du chapitre 3. Avant d'y être, détaillons maintenant la méthode de protection introduite dans le *benchmark* ainsi que la méthode de simulation d'injection de scénarii de fautes dans l'émulateur.

# C. Développement d'outils d'aide à l'évaluation de la sûreté de fonctionnement

## C.1. Nécessité de compromis de sûreté logicielle/matérielle

Comme nous venons de le voir dans le paragraphe précédent, l'ensemble émulateur/*benchmark* nous sert à affiner l'architecture, en particulier dans le choix du nombre de bus. De plus, il nous sera possible d'intégrer des modèles d'injection d'erreurs dans l'émulateur ainsi que des techniques de protection logicielles dans le *benchmark*.

En effet, les techniques de protection du processeur peuvent être soit logicielles soit matérielles (Figure 2.9). Nous évoquons ici des techniques traitant les erreurs susceptibles de se produire dans le processeur et non pas dans l'application. Intéressons-nous aux

conséquences de ces techniques : d'un coté, si nous tranchons pour les techniques matérielles, ceci engendrera un surcoût matériel dû à l'ajout du matériel susceptible d'allonger le chemin critique. Cela peut même influencer sur les performances temporelles. D'un autre côté, si nous tranchons pour les techniques logicielles, il y aura une baisse des performances temporelles due à l'ajout de nouvelles routines dédiées à la protection, ce qui nécessite d'allonger le programme.

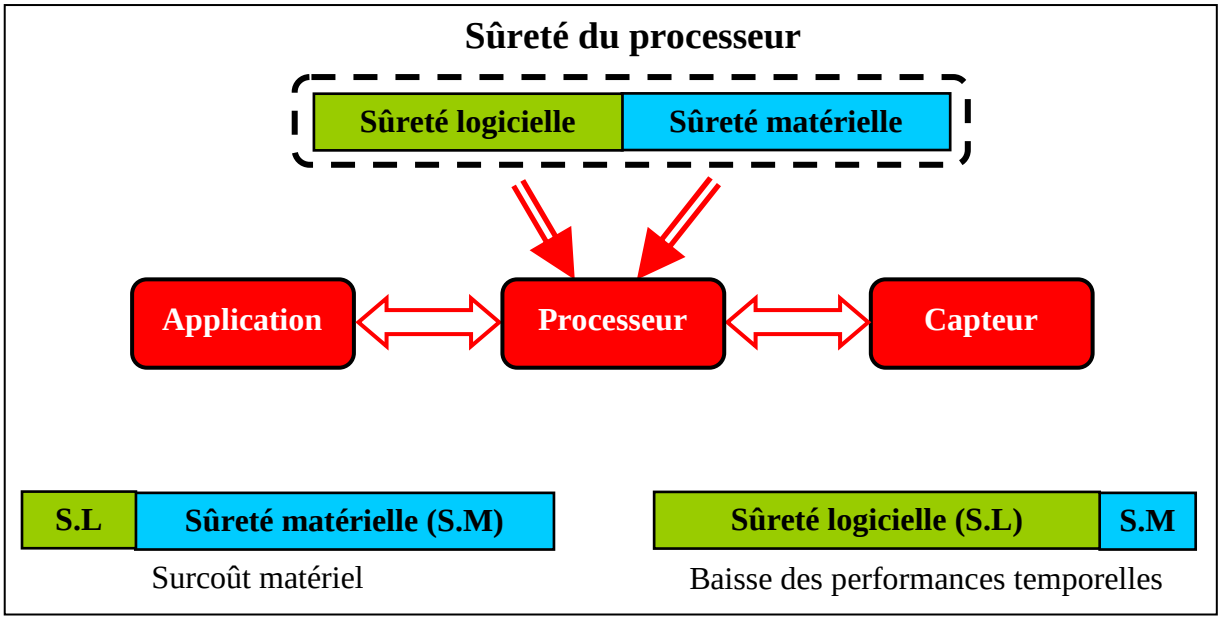


**Figure 2.9 : Nécessité de compromis de sûreté logicielle/matérielle**

Lors de l'utilisation des techniques logicielles et matérielles, plusieurs cas peuvent se présenter. Nous citons, entre autres :

- La routine de protection logicielle suffit amplement, c'est-à-dire que le taux d'erreurs est faible et/ou les registres sont entièrement protégés matériellement.
- La protection logicielle est insuffisante, c'est-à-dire que le logiciel n'a même plus le temps de traiter une erreur qu'une seconde erreur arrive ce qui nous oblige à réduire ce taux d'erreurs avec des techniques matérielles.
- La protection matérielle est insuffisante, certaines erreurs détectées seront ainsi non corrigées matériellement ce qui a pour conséquence de faire 'sous-traiter' ces erreurs par le logiciel.

Ainsi, pour une application donnée située dans un environnement particulier, et en connaissant certains critères tels que le taux d'erreurs temporel, les contraintes temporelles de l'application, ou aussi des contraintes de puissance (c'est-à-dire des contraintes matérielles), nous pouvons avoir une estimation qui nous précise jusqu'à quel niveau nous pouvons utiliser

de telles méthodes de protection. Un compromis entre les deux types de protection s'avère alors nécessaire. C'est pour cette raison que nous allons exploiter l'ensemble émulateur/*benchmark* afin d'identifier la limite de protection logicielle/matérielle à travers une technique de protection logicielle, objet de la partie suivante.

## C.2. Principe de la méthode de protection introduite dans le Benchmark

Cette technique de protection est résumée dans la figure 2.10. Le principe consiste à injecter des erreurs dans l'émulateur. En supposant qu'il existe un mécanisme de détection matérielle d'erreurs, nous provoquons une interruption à chaque erreur détectée et non corrigée par le matériel. Donc, une routine est ajoutée dans l'émulateur pour générer une interruption toutes les N instructions, N pouvant être constant ou aléatoire. Le nombre d'interruptions peut également être constant ou aléatoire. Suite aux conséquences de cette injection d'erreurs, une technique de protection logicielle est implantée dans le *benchmark* de tri. Elle consiste à faire un retour au dernier état sûr. Ceci se fait en ajoutant dans le *benchmark* des fonctions de sauvegarde et de lecture des sommets des piles, du compteur programme et des pointeurs de piles (sauvegarde partielle) avec les données traitées (sauvegarde complète). La sauvegarde du contexte est périodique alors que le retour au dernier état sûr est réalisé à chaque interruption.

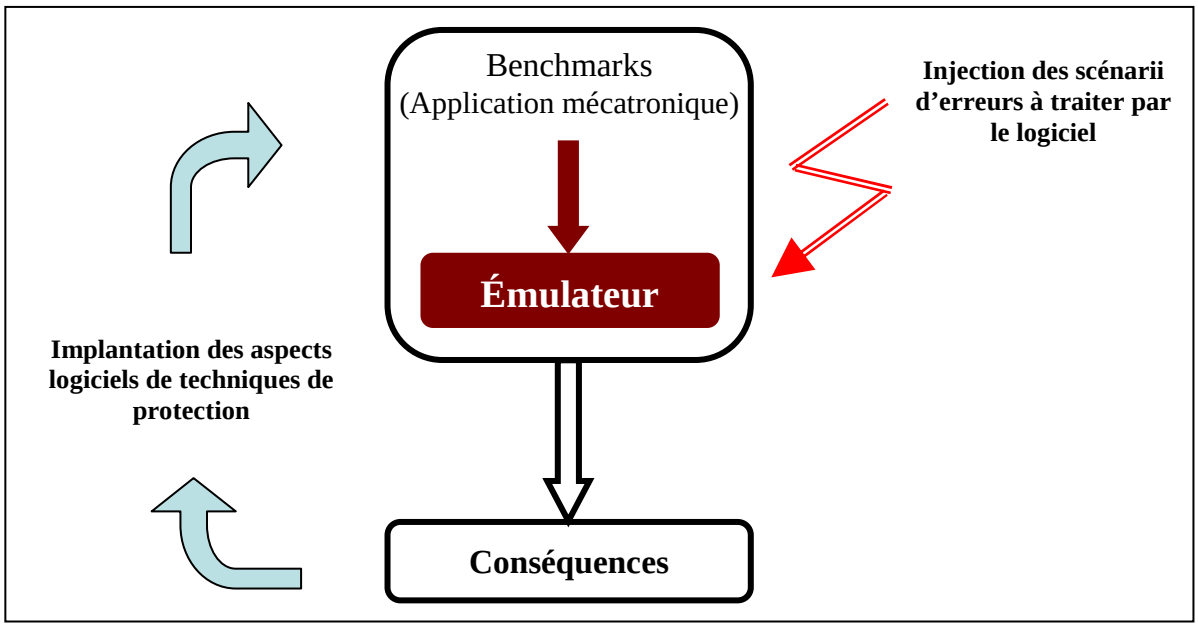


**Figure 2.10 : Utilité de l'ensemble émulateur/benchmark pour l'intégration des injections de fautes et des méthodes de protection**

C'est pour avoir la possibilité de faire la sauvegarde et la restauration qu'un ajout de quatre instructions a été nécessaire (Tableau 2.4, section B.5) afin de placer les contenus des

pointeurs de piles au sommet de la pile. Rappelons-les : Push_DSP (sauvegarder DSP dans le TOS), Push_RSP, Pop_DSP (restaurer DSP du TOS), Pop_RSP. Déjà, nous remarquons que cet ajout d'instructions a un effet sur le matériel, ce dont nous ne pouvions pas nous rendre compte sans l'utilisation de l'émulateur. La méthode de protection et la technique d'injection de fautes étant définies, détaillons maintenant l'émulateur et les *benchmarks* développés.

## C.3. Développement de l'émulateur

Actuellement, l'émulateur permet de lire un programme qui est sous forme de fichier hexadécimal et de le stocker dans la zone mémoire appropriée (partie chargeur : '*loader*'). Suite à l'exécution de ce programme, il décrit les changements affectant la mémoire et les deux piles (partie exécuteur). Cette structure interne de l'émulateur est décrite dans la Figure 2.11.

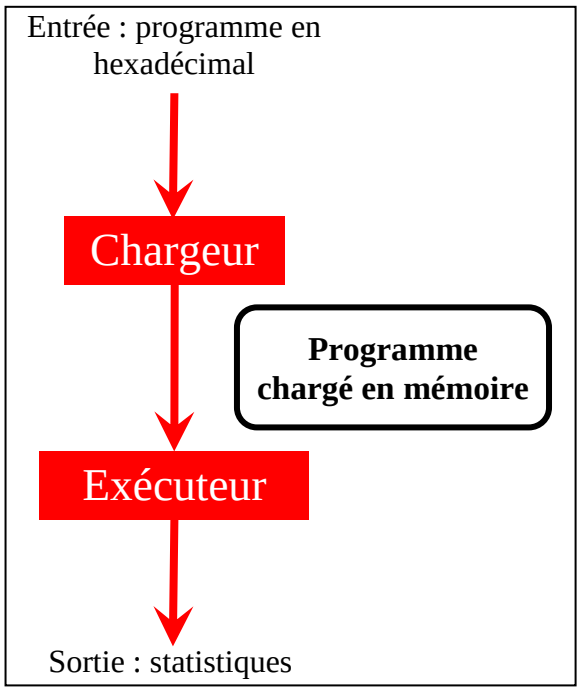


**Figure 2.11 : Structure globale de l'émulateur**

L'architecture étant définie, 38 instructions sont proposées. En se basant sur l'analyse des conséquences de chaque instruction sur les piles et les mémoires, un émulateur est développé en langage C.

Initialement un fichier assembleur représentatif d'un programme embarqué dans le processeur, est converti en hexadécimal. Il est chargé en mémoire : les codes hexadécimaux se trouvent ainsi dans la mémoire. La partie exécuteur exécute ces codes un par un et effectue

les changements affectant les piles et les mémoires, comme il est détaillé dans la Figure 2.12 suivante.

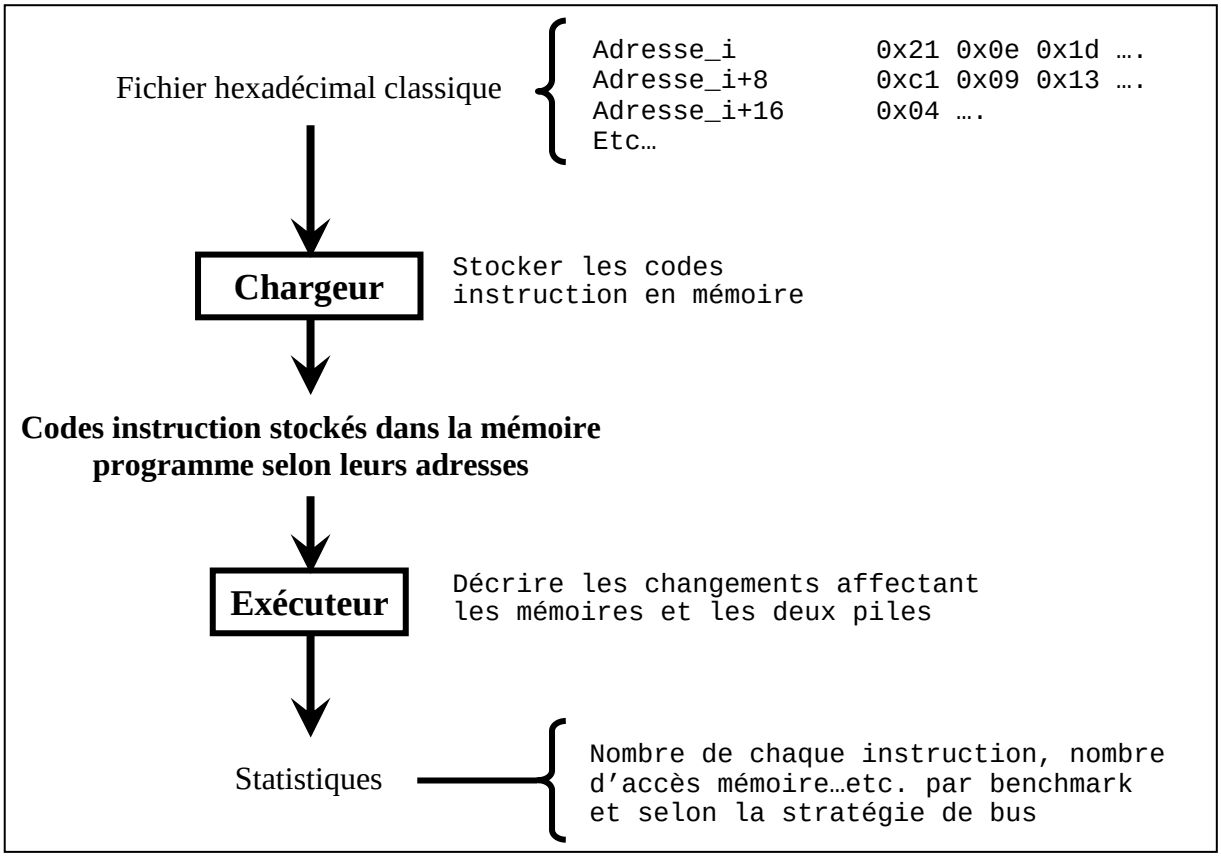


**Figure 2.12 : Structure détaillée de l'émulateur**

Concernant la structure de données, la machine virtuelle est composée du compteur programme et des sommets et des pointeurs des deux piles. Nous avons défini des compteurs des instructions et des compteurs pour les différents accès mémoire. Nous donnons dans la suite les deux algorithmes du chargeur et de l'exécuteur.

- **Algorithme du chargeur :** Lecture du programme (qui est sous forme de fichier hexadécimal) et stockage dans la zone mémoire appropriée

```
Ouverture et lecture du fichier hexadécimal
Tant que (non fin du fichier)
      {
      Pour chaque ligne du fichier
            {
            Extraction adresse
            Extraction des codes instruction
```

```
            Stockage des codes dans les zones appropriées de la
            mémoire (selon l'adresse extraite)
            }
      }
```

- **Algorithme de l'exécuteur :** Description des changements affectant les mémoires et les deux piles suite à l'exécution du programme

```
Tant que (non fin du programme)
{
Récupérer l'adresse du compteur programme (PC)
Récupérer le code instruction dans la mémoire (qui est d'adresse le
contenu du PC)
Selon l'instruction
      {
      Mise à jour des zones mémoires nécessaires :
            - Piles uniquement
            - Piles et mémoire
            - Mémoire uniquement (STORE)
            - Compteur programme (BRA : branchement)
      Mise à jour des compteurs d'instructions (selon la stratégie
      bus)
      Mise à jour des compteurs des différents accès mémoire
      }
Incrémentation du compteur programme
}
```

# D. Conclusions

Dans ce chapitre, nous avons défini notre démarche adoptée en vue d'une méthodologie de conception d'architecture de processeur sûre. Cette démarche est principalement basée sur le développement d'un émulateur de processeur que nous avons montré son utilité dans un objectif de sûreté de fonctionnement. En effet, nous avons proposé une technique de protection qui se base sur le principe de reprise (recouvrement arrière) en tant que moyen pour la sûreté. Cette technique, implantée dans le *benchmark*, est basée sur la sauvegarde périodique des éléments constitutifs du cœur du processeur. Le principe consiste à générer des interruptions dans l'émulateur correspondantes à une injection d'erreurs. À chaque interruption, l'exécution de l'application est momentanément arrêtée et la routine de protection prend la relève en retournant au dernier état sûr. Une fois revenue au dernier état

sûr, l’application reprend son exécution. Evidemment, l’émulateur ne sera se développer sans définir et justifier le type d’architecture ainsi que sa structure interne et son jeu d’instructions. C’est pour cela, une étude préalable et comparative a été établie afin de justifier notre démarche qui a abouti au choix d’un processeur 0-opérande à deux piles. Dans le chapitre suivant, nous allons exploiter l’ensemble émulateur/*benchmark* afin de faire le choix sur le nombre de bus d’adressage mémoire d’une part et de quantifier les capacités de correction de la méthode de protection définie dans ce second chapitre d’autre part.

# Références


— [DILG03] E, Dilger, M, Gulbins, T, Ohnesorge and B, Straube “On a Redundant Diversified Steering Angle Sensor”, 9th IEEE International On-Line Testing Symposium (IOLTS’03), pp. 191- 196, July 2003.

— [DILG04] E. Dilger, R. Karrelmeyer and B. Straube “Fault tolerant mechatronics”, in Proc. 10th IEEE International On-Line Testing Symposium, 2004 (IOLTS’04), pp. 214- 218, July 2004.

— [ISER02] R. Isermann, R. Schwarz and S. Stolzl “Fault-tolerant drive-by-wire systems”, in IEEE Control Systems Magazine, vol. 22, issue 5, pp. 64- 81, October 2002.

— [KOCH04] H. D. Kochs: “Key Factors Key dependability of mechatronic units”, 28th IEEE Annual International Computer Software and Application Conference (COMPSAC’04), September 2004.

— [KOOP89] P.J. Koopman, Jr.: “Stack Computers: The New Wave”, California: Ed. Mountain View Press, 1989;
http://www.ece.cmu.edu/~koopman/stack_computers/.

— [MOOR95] C. Moore and C.H. Ting: “Minimal Instruction Set Computer”, Forth Dimensions, January 1995.

— [SHAO03] A. Shaout and T. Eldos: “On the classification of computer architecture”, International Journal of Science and Technology, vol. 14, USA 2003.

— [SPRI01] P. Springer: “Assessing Application Vulnerability to Radiation-Induced SEUs in Memory”, NASA Technical Reports Server, 2001.

— [TING95] C. Ting and C. Moore: “Mup21: a high performance MISC processor”, Forth Dimensions, January 1995.

# Chapitre 3

# Étude quantitative : mise en œuvre et simulation

Maintenant que les approches sont définies et justifiées et que les outils d'aide à l'évaluation de la sûreté de fonctionnement sont développés, nous procédons à la partie quantification. En effet, nous exploitons l'ensemble émulateur/*benchmark* afin de quantifier les performances temporelles des deux stratégies bus. Nous les comparons et nous vérifions leur influence sur la sûreté de fonctionnement, ce que nous abordons dans ce chapitre dans le but de fixer le nombre de bus. Ceci concerne l'affinement de l'architecture. Quant au mécanisme de protection, nous évaluons la borne de protection logicielle/matérielle pour différents scénarii d'injection d'erreurs. Il s'agira de quantifier les capacités de corrections d'erreurs de la technique de protection et d'évaluer son surcoût temporel. En effet, supposons que nous disposons d'un cahier de charges défini par la durée d'exécution normale sans apparition d'erreurs d'une application donnée, et sa contrainte temps-réel qui consiste à ne pas dépasser 130% par exemple. Notre objectif est de trouver une méthode applicable à n'importe quel processeur nous permettant de déterminer à partir de quel critère le processeur ne remplit plus son service.

# A. Choix de la stratégie de bus

Comme nous l'avions précédemment décrit, concernant le nombre de bus d'adressage du processeur, celui-ci peut être adressé à travers deux ou trois bus. Nous avons alors développé des *benchmarks* afin de comparer les performances temporelles de ces deux stratégies différentes dans une première étape et de conclure sur le nombre de bus d'adressage à adopter. Ces *benchmarks* nous seront aussi d'une grande utilité dans la suite puisque nous allons y intégrer directement notre méthode de correction. Ce qui nous permettra dans une seconde étape d'évaluer et de valider cette technique de correction.

## A.1. Développement de benchmarks

À l'état actuel, trois différents *benchmarks* sont développés pour l'analyse et l'évaluation des performances :

- ***Benchmark1* :** calcul d'équations logiques et arithmétiques dont les entrées, en provenance du capteur, et préalablement stockées dans la mémoire, sont traitées selon ces équations. Les sorties sont ensuite enregistrées dans la mémoire pour être envoyées aux actionneurs.
- ***Benchmark2* :** permutation deux à deux de dix variables stockées en mémoire.
- ***Benchmark3_Condition_initiale_i* :** tri à bulles de dix variables stockées en mémoire avec cinq différentes conditions initiales.

À noter que nous avons pris le choix de développer de tels benchmark tout en exploitant la très grande majorité des 38 instructions du jeu du processeur. Dans le *benchmark1*, nous supposons que nous avons un système à commander qui demande un calcul de paramètres de correcteurs numériques et par la suite les sorties via de diverses équations arithmétiques. En outre, nous supposons qu'il dispose d'entrées/sorties tout ou rien, ce qui peut correspondre à un ensemble d'équations logiques. Concernant le *benchmark2*, comme nous avons voulu réaliser un *benchmark* plus complexe que le premier, nous nous sommes orientés vers des algorithmes basiques tels que le tri, qui n'est autre que le *benchmark3*. Nous avons fait plusieurs versions de ce dernier en modifiant cinq fois les conditions initiales (les valeurs et les adresses des données à trier). Remarquant que pour aboutir au *benchmark3*, il nous a fallu le développement d'une routine qui permute deux données, nous l'avons alors exploité ainsi que les dix données initialement destinées à être trier, pour en sortir un nouveau *benchmark2* dont le rôle est la permutation deux à deux de dix variables présentes en mémoire. Ceci dit, rien ne nous empêche de développer bien entendu d'autres *benchmarks*.

- ***Benchmark1* :**

Nous nous proposons d'implanter un code assembleur calculant des équations logiques et arithmétiques dont les entrées, en provenance du capteur, et préalablement stockées dans la mémoire, sont traitées selon ces équations. Les sorties sont ensuite enregistrées dans la mémoire pour être envoyées aux actionneurs. Nous avons choisi de telles fonctions pour le *benchmark1* dans la mesure où dans certains systèmes industriels, pour le calcul de commandes des actionneurs, auront besoin de calculer des équations de ce genre.

Hypothèses :

- Les mesures en provenance des capteurs sont stockées en mémoire (exemple : dix mesures de $x_0$ à $x_9$).
- Les commandes à calculer et à envoyer aux actionneurs seront aussi stockées en mémoire (exemple : quatre commandes de $d_0$ à $d_3$).
- Les équations de commande sont les suivantes :

$\mathbf{d_0}$ = constante$_1$ x [($x_0$+$x_1$) - ($x_2$ modulo $x_3$)] / [$x_4$ x (-$x_5$)] + cte$_2$

$\mathbf{d_1}$ = NOT [($x_6$ OR $x_1$) AND ($x_9$ XOR $x_7$)] AND [NOT($x_8$)]

***if* ((x8+x9)<cte3)**

$\mathbf{d_2}$ = constante$_3$ x [($d_0$+$x_1$) - ($x_9$ x $x_8$)] DIV [$x_1$ - $x_5$] + (constante$_4$ modulo $d_0$)

***else***

$\mathbf{d_2}$ = [($d_0$ + $x_1$)-($x_9$ x $x_8$)] DIV [$x_1$ - $x_5$] + (constante$_4$ modulo $d_0$)

$\mathbf{d_3}$ = NOT [($x_6$ OR $d_1$) AND ($x_9$ XOR $x_7$)] AND [NOT($d_1$)]

Afin de l'exécuter sur un processeur à pile de données, nous devions suivre une succession d'étapes bien déterminée. Prenons l'exemple de la première équation :

$d_0$ = constante$_1$ x [($x_0$+$x_1$) - ($x_2$ modulo $x_3$)] / [$x_4$ x (-$x_5$)] + cte$_2$

Le code assembleur lui correspondant est alors le suivant, accompagné des commentaires nécessaires :

```
//Étape 1
DLIT 0xe000
FETCH                                   //xo (d'adresse 0xe000) est sur le sommet
DLIT 0xe002
```

```
FETCH                                          //x1 (d'adresse 0xe002) est sur le sommet
ADD                                                                  //TOS = x0 + x1 ;
//Étape 2
DLIT 0xe004
FETCH                                          //x2 (d'adresse 0xe004) est sur le sommet
DLIT 0xe006
FETCH                                          //x3 (d'adresse 0xe006) est sur le sommet
MOD                                                    //TOS= x2 % x3 ; NOS = x0 + x1
//Étape 3
SUB                                                          // TOS = (x0 + x1) - (x2 % x3) ;
//Étape 4
DLIT 0xe008
FETCH                                          //x4 (d'adresse 0xe008) est sur le sommet
DLIT 0xe00a
FETCH                                          //x5 (d'adresse 0xe00a) est sur le sommet
NEG                                                                  // (-x5) est sur le sommet
MUL                                    //TOS =  (x4 x (-x5)) ; NOS = (x0 + x1) - (x2 % x3)
//Étape 5
DIV                                    //TOS = [(x0 + x1) - (x2 % x3)] / [(x4 x (-x5))] ;
//Étape 6
DLIT constante1
MUL                            //TOS = constante1 x [(x0 + x1) - (x2 % x3)] / [(x4 x (-x5))] ;
//Étape 7
DLIT constante2
ADD          //TOS = constante1 x [(x0 + x1) - (x2 % x3)] / [(x4 x (-x5))] + constante2 (= d0)
DLIT 0xd000                  // l'adresse ou va être stockée le résultat de la commande d0
                                                                  // TOS = 0xd000 ; NOS = d0
STORE                          // la valeur calculée (d0) est stockée dans l'adresse 0xd000
```

L'ordre de calcul peut alors être schématisé par la Figure 3.1 suivante :

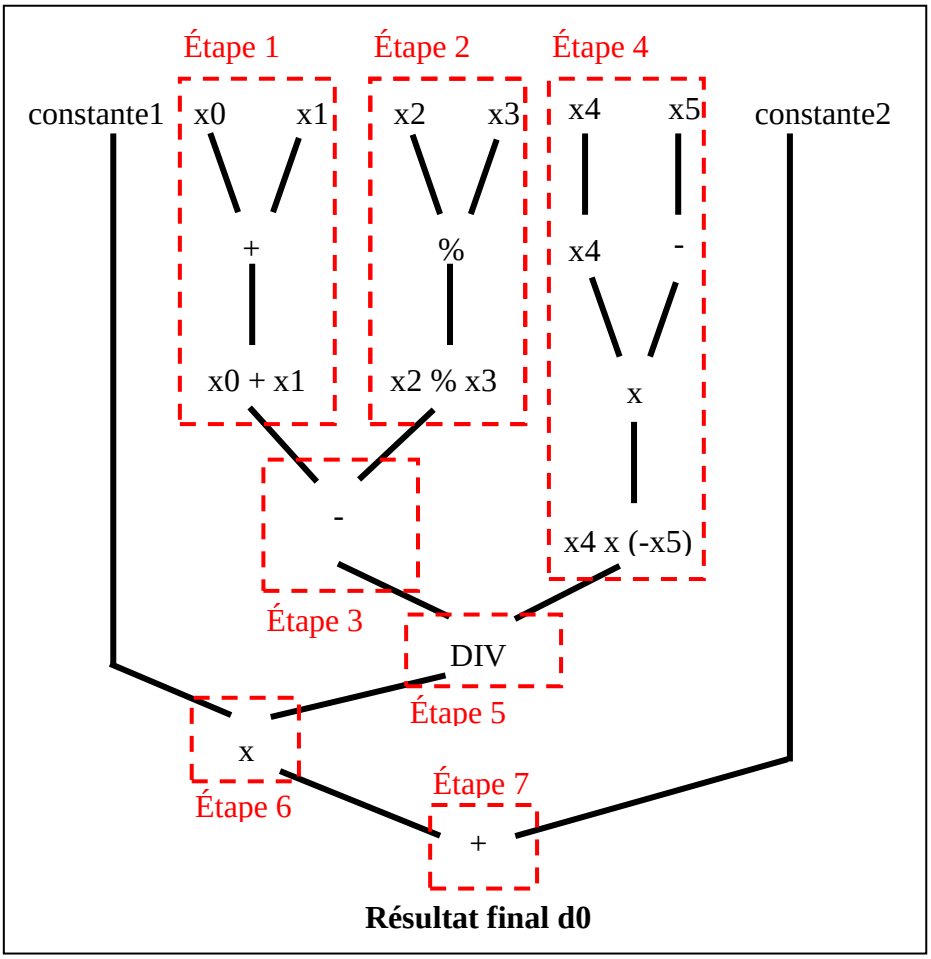


**Figure 3.1 : Enchaînement des étapes de calcul lors d'une opération arithmétique s'exécutant sur un processeur à deux piles**

Le traitement est de même pour les trois autres équations. Ce qui nous fait une totalité de plus d'une centaine d'instructions. Le code assembleur est ensuite converti en fichier hexadécimal pour obtenir 212 octets (supérieur à la centaine d'instructions dans la mesure où des littéraux sont ajoutés dans le programme).

- ***Benchmark2 :***

Nous avons une liste de dix variables que nous voulons permuter. Nous gardons la même hypothèse que pour le *benchmark1* : les mesures sont stockées en mémoire (exemple : dix mesures de $x_0$ à $x_9$). Nous fixons comme condition initiale : chaque valeur ($x_i$) est stockée dans l'adresse (`adresse_`$x_i$). Nous avons choisi de telles fonctions pour le *benchmark2* dans la mesure où pour le prochain *benchmark3*, nous aurons besoin de sous-fonctions de permutation. Une partie du code assembleur de ce *benchmark2* est décrite et commentée dans la suite. Elle correspond à la permutation des données (`x0`) et (`x1`) et est suivie par une description détaillée.

```
DLIT adresse_x0                                  // TOS = adresse_x0
FETCH                                            // TOS = x0
DLIT adresse_x1                                  // TOS = adresse_x1 et NOS = x0
FETCH                                            // TOS = x1 ; NOS = x0
DLIT adresse_x0                  // TOS = adresse_x0 ;  NOS = x1 ; 3ème élément = x0
DLIT adresse_x1                                  // TOS = adresse_x1 ; NOS = adresse_x0
                                                 //3ème élément = x1 ; 4ème élément = x0
D2R                              // TOS = adresse_x0 ; NOS = x1 ; 3ème élément =  x0
                                                 // TORS = adresse_x1
STORE                            // TOS = adresse_x0 ; x1 est stockée dans adresse_x0
                                                 // TOS = x0
                                                 // TORS = adresse_x1
R2D                                              // TOS = adresse_x1 ; NOS =  x0
STORE                                            // x0 est stockée dans adresse_x1
```

Le principe consiste à empiler tout d'abord la première adresse de la valeur que nous voulons permuter grâce à l'instruction (`DLIT adresse_x0`). Ensuite, pour pouvoir récupérer la valeur stockée dans cette adresse, nous utilisons l'instruction (`FETCH`). Nous faisons de même pour (`x1`). Ainsi, nous avons (`x1`) et (`x0`) respectivement dans le TOS et le NOS. L'instruction (`STORE`) requiert les nouvelles adresses afin d'y écrire les nouvelles valeurs. Pour cela, nous chargeons de nouveau (`adresse_x0`) et (`adresse_x1`) successivement, de telle manière nous obtenons (`TOS = adresse_x1`), (`NOS = adresse_x0`), (3ème élément = `x1`) et (4ème élément = `x0`). C'est à ce moment là que nous avons besoin de la pile de retour pour l'exploiter dans une copie temporaire. En effet, pour sauvegarder une nouvelle donnée dans une adresse bien déterminée avec l'instruction (`STORE`), il faut avoir l'adresse dans le TOS et la donnée dans le NOS. C'est pour cette raison que nous utilisons l'instruction (`D2R`) nous permettant de déplacer le sommet de la pile de donnée (`adresse_x1`) vers le sommet de la pile de retour. Ainsi, nous aurons respectivement (`adresse_x0 x1 x0`) dans les trois premiers éléments de la pile de données. En faisant un (`STORE`), la donnée (`x1`) est désormais stockée dans l'adresse (`adresse_x0`). En récupérant la donnée (`adresse_x1`) de la pile de retour à travers l'instruction inverse (`R2D`), nous aurons respectivement (`adresse_x1 x0`) dans le TOS et le NOS. De même, avec un (`STORE`), la donnée (`x0`) est désormais stockée dans l'adresse (`adresse_x1`). D'où la fin de l'opération de permutation de (`x0`) et (`x1`). Le traitement est identique pour le reste des données.

- ***Benchmark3 :***

Nous avons une liste de dix variables que nous voulons ordonner. Nous gardons la même hypothèse que précédemment : les mesures sont stockées en mémoire (exemple : dix mesures de $x_0$ à $x_9$). Nous fixons comme condition initiale : chaque valeur $x_i$ est stockée dans l'adresse adresse_$x_i$. L'algorithme est expliqué dans la figure suivante :

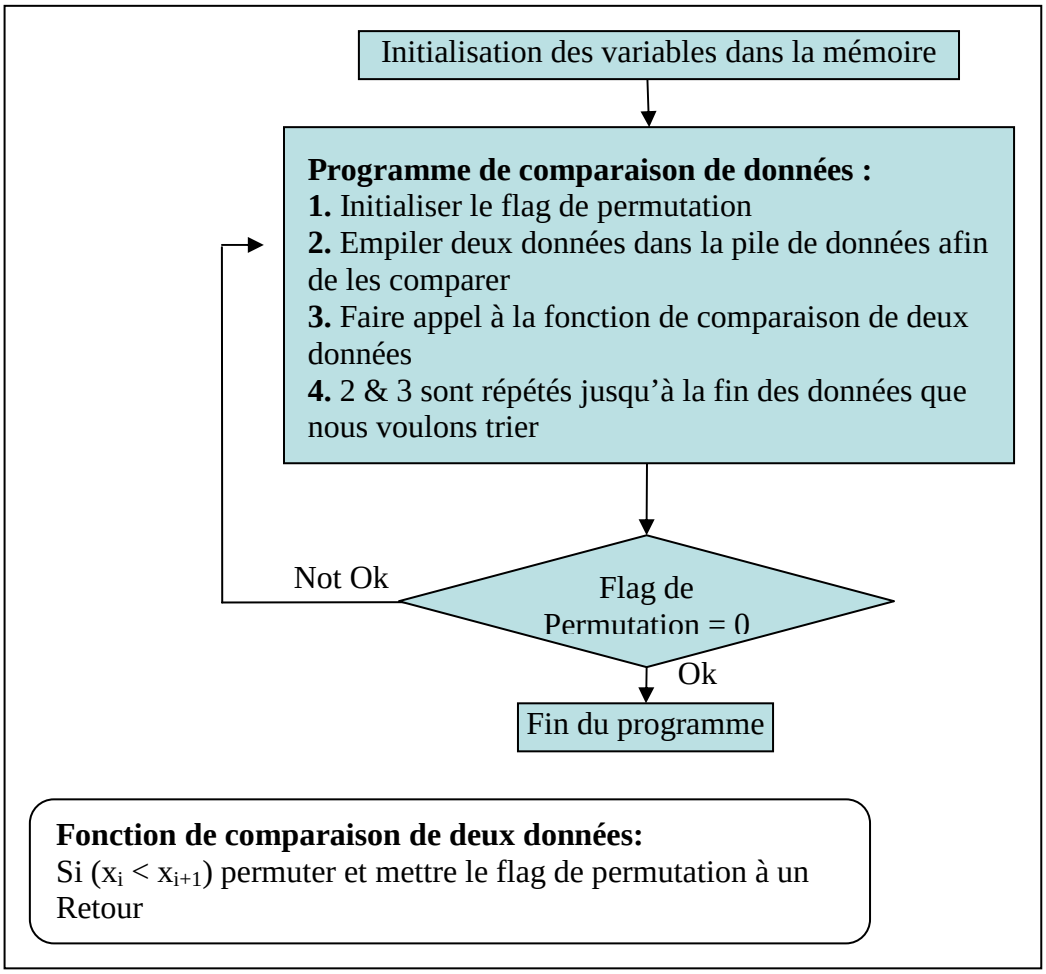


**Figure 3.2 : Algorithme de tri à bulles sans prise en compte de la tolérance aux fautes**

Nous présentons à titre d'exemple le code assembleur de la fonction de comparaison de deux données qui n'est autre qu'une sous-fonction du *benchmark3* qui traite un tri croissant.

```
fct_comparaison_2_données:
                                    //condition initiale : TOS = data1 et NOS = data2
SUB                                 // TOS = data2-data1
SIGNE                               //Si (data2>=data1) Alors TOS ← 0x00 sinon TOS ← 0xFF
ZBRA 0x04                           //Si (TOS=0) Alors Saut de 4 octets (pas de permutation),
                                    // Sinon pas de saut (permutation des deux données)
```

```
SWAP                    // permuter les deux données ➔ TOS = data2 et NOS = data1
DLIT 0x0001                                                    // Charger un 1
                                        //TOS = 1, NOS = data2 et 3ème élément = data1
DLIT 0xC000                                  // Charger l'adresse du flag de permutation
                  //TOS = 0xC000, NOS = 1, 3ème élément = data2 et 4ème élément = data1
STORE                  // MEM [0xC000] reçoit un.  (flag de permutation devient égal à 1)
RETURN
```

Après avoir chargé les deux données (`data1`) et (`data2`) de la mémoire et les placer respectivement dans le NOS et le TOS, nous les comparons en utilisant l'opération soustraction. Grâce à l'instruction (`SIGNE`), nous pouvons déterminer le supérieur de ces deux données. Si `data2 – data1 >= 0` (donc `data1 =< data2`) Alors TOS = 0x0000. Les deux données sont bien ordonnées et n'ont pas besoin d'une permutation. Sinon le résultat de la soustraction est négatif et par conséquent le SIGNE d'un TOS négatif devient un 0xFFFF. À l'aide de l'instruction (`ZBRA 0x04`), un saut de quatre octets s'effectue si le TOS est nul. S'il n'est pas nul (le résultat de soustraction est négatif) alors le saut ne s'effectue pas et la suite des instructions est exécutée, à savoir la permutation et la mise à un du flag de permutation.
Maintenant que nous avons défini les *benchmarks*, nous les exploitons pour l'étude quantitative et comparative des performances temporelles des deux stratégies mémoires proposées.

## A.2. Exploitation des benchmarks dans le choix de nombre de bus

Nous donnons dans le Tableau 3.1 suivant, le regroupement des instructions selon leurs accès mémoire :

| Besoin d'accès aux (à la) : | **Nombre d'instructions :** |
|---|---|
| Mémoire programme + Mémoire pile de données | 29 |
| Mémoire programme + Mémoire pile de retour | 2 (CALL, RETURN) |
| Mémoire programme | 2 (HALT, NOP) |
| Mémoire programme + Mémoire pile de données + Mémoire pile de retour | 3 (R2D, D2R, CPR2D) |
| Mémoire programme + Mémoire pile de données + Mémoire explicite | 2 (FETCH, STORE) |

**Tableau 3.1 : Regroupement des instructions selon leurs accès mémoire**

Nous constatons que parmi 38 instructions, cinq uniquement ont besoin d'accéder à trois zones mémoires différentes pour pouvoir s'exécuter. D'ailleurs, notons que nous nous sommes éloignés de la stratégie un bus dès le départ dans la mesure où dans un tel cas, quasiment toutes les instructions seront retardées à part HALT et NOP. Dans le cas d'une stratégie deux bus, le plus convenable est de mettre chacune de la mémoire pile de données et de la mémoire programme sur un bus différent. En effet, 34 instructions sur 38 utilisent ces deux zones mémoires parmi eux, cinq instructions utilisent une troisième zone mémoire. Si nous ne les séparons pas, au minimum 34 instructions seront retardés et contraintes d'attendre un cycle supplémentaire pour la libération de leur éventuel bus commun. Dans ce cas et avec séparation des accès à ces deux mémoires, trois solutions sont envisageables :

**Stratégie 2 bus**

- ***Solution 1 :***

  Bus1 : Mémoire programme
  Bus2 : Mémoire pile de données + Mémoire pile de retour + Mémoire explicite

- ***Solution 2 :***

  Bus1 : Mémoire programme + Mémoire explicite
  Bus2 : Mémoire pile de données + Mémoire pile de retour

- ***Solution 3 :***

  Bus1 : Mémoire programme + Mémoire pile de retour
  Bus2 : Mémoire pile de données + Mémoire explicite

Si nous regardons bien la solution 3, nous trouvons que cette solution est la moins bonne entre les trois solutions. En effet, bien que nous sommes en stratégie deux bus, et que les instructions CALL et RETURN n'utilisent que deux zones mémoires : la mémoire programme et la mémoire pile de retour (Tableau 3.1), ces deux instructions se voient retardés d'un cycle supplémentaire dans la mesure où ces deux mémoires sont adressées par un même bus, ce qui n'est pas le cas des solutions 1 et 2. Ces deux solutions ne retarderont en aucun cas l'exécution des instructions CALL et RETURN. Par conséquent, la solution 3 est à retirer. Ne nous restant que les alternatives 1 et 2, et remarquant que pour ces deux situations, nous avons toujours cinq instructions (R2D, D2R, CPR2D, FETCH et STORE) qui s'exécutent en deux cycles, nous choisissons la solution 1 pour une raison d'homogénéité. En effet, nous préférons garder la mémoire programme dans un seul bloc à part et regrouper les autres mémoires, contenant des données, dans un autre. Ainsi la solution retenue est la solution 1 de la stratégie deux bus :

**Stratégie 2 bus**

- ***Solution 1 :***

Bus1 : Mémoire programme
Bus2 : Mémoire pile de données + Mémoire pile de retour + Mémoire explicite

Regardons le cas de la stratégie trois bus et discutons les solutions envisageables. Tout d'abord, il nous faut séparer la mémoire pile de données et la mémoire programme pour les mêmes raisons évoquées ci-dessus. Ensuite, un simple regard sur le Tableau 3.1 nous permet de remarquer qu'aucune instruction n'utilise à la fois ces deux mémoires : la mémoire explicite et la mémoire pile de retour. C'est pour cette raison que nous décidons de regrouper ces deux mémoires dans un même bloc de telle manière qu'elles soient adressées par le même bus. Ainsi, avec une telle stratégie, nous n'aurons aucun retard dû aux accès multiples à un seul bus pour l'exécution d'une instruction. Ces accès sont donnés par :

**Stratégie 3 bus**

Bus1 : Mémoire programme
Bus2 : Mémoire pile de données
Bus3 : Mémoire pile de retour + Mémoire explicite

À partir de là, des compteurs sont ajoutés dans le code de l'émulateur afin de calculer l'accès à chacune des mémoires pour ensuite établir des statistiques sur les différents accès pour chaque *benchmark.* Pour le *benchmark3,* nous avons fait l'étude pour cinq conditions initiales différentes, en modifiant les adresses et les valeurs des données à trier. Nous relevons alors la perte en performances temporelles dû par un accès deux bus. Une comparaison entre les deux différentes stratégies de bus est illustrée dans le Tableau 3.2 suivant :

| | **Perte de performances temporelles 2bus par rapport à 3bus** |
|---|---|
| ***Benchmark 1*** | **31%** |
| ***Benchmark 2*** | **35%** |
| ***Benchmark 3*** **Condition Initiale 1** | **30%** |
| ***Benchmark 3*** **Condition Initiale 2** | **28%** |
| ***Benchmark 3*** **Condition Initiale 3** | **28%** |
| ***Benchmark 3*** **Condition Initiale 4** | **27%** |
| ***Benchmark 3*** **Condition Initiale 5** | **28%** |
| **Moyenne** | **30%** |

**Tableau 3.2 : Perte de performances temporelles deux bus par rapport à trois bus**

Comme nous le constatons, avec une stratégie deux bus, nous avons une durée d'exécution 30% plus longue en moyenne. Cette perte en temps peut avoir une conséquence sur la sûreté du processeur. En effet, supposons que le processeur est jugé sûr pour un taux d'erreurs de 1 erreur toutes les 7 ms, et que le programme qui y tourne dure 6 ms en stratégie trois bus et par conséquent 8ms (30% de plus) en stratégie deux bus. Un environnement à 1 erreur toutes les 7 ms aura pour conséquence la non sûreté du processeur avec une stratégie en deux bus. Dans ce cas, la stratégie trois bus est meilleure. De plus, à part cette perte en performances temporelles à cause de la stratégie deux bus, nous risquons d'avoir d'autres pertes en utilisant des méthodes de protection logicielles, ce qui sera démontré dans la partie B suivante de ce chapitre. En fait, ces techniques logicielles peuvent ralentir le programme puisque certaines routines de protection seront intégrées dans le *benchmark*.

# B. Évaluation de la méthode de protection

## B.1. Intégration des routines de protection

Nous ajoutons dans le *benchmark* deux routines dédiées à la protection. Une qui fait la sauvegarde périodique des éléments constituant le cœur du processeur, à savoir les sommets des piles, le compteur programme et les pointeurs de piles dans une mémoire externe supposée sûre. Au cours de l'exécution normale de l'application et dès qu'une erreur apparaît, une interruption est alors déclenchée, l'application est interrompue le temps d'exécuter la routine de recouvrement arrière. En effet, cette dernière restaure le dernier état sûr, à savoir restaurer les derniers éléments sauvegardés dans un objectif de reprendre les traitements à partir de ce dernier état sûr. Nous donnons alors le nouvel algorithme de tri à bulles avec prise en compte de la tolérance aux fautes.

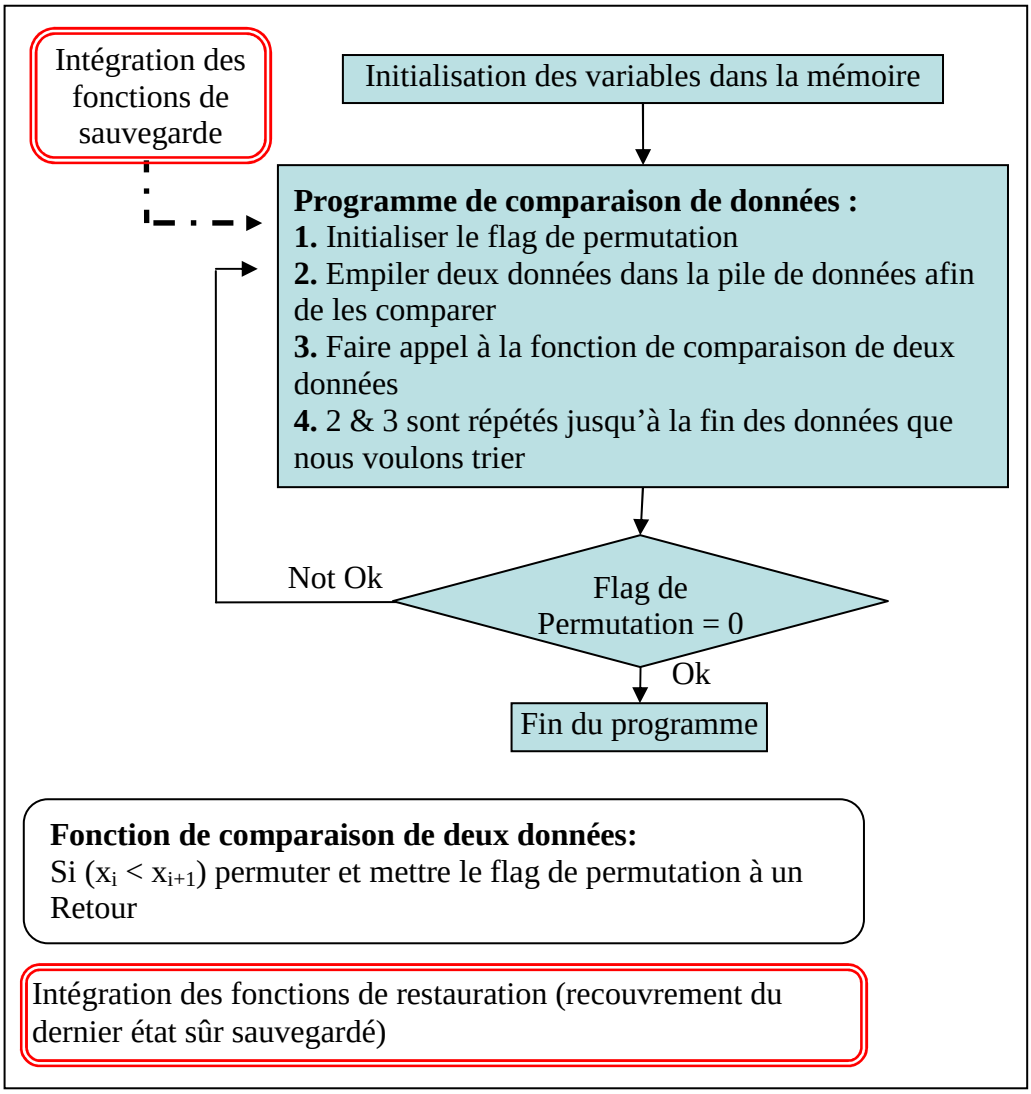


**Figure 3.3 : Algorithme de tri à bulles avec prise en compte de la tolérance aux fautes**

Le principe est de faire une sauvegarde périodique. Le retour au dernier sûr se fait par la lecture de ce qui a été déjà sauvegardé, donc aller chercher les données dans les adresses ou elles étaient sauvegardées et de les remettre à leur place. Ces fonctions ajoutées sont des fonctions indépendantes de l'application. Elles peuvent s'intégrer dans un autre *benchmark*. Il suffit juste de changer les adresses de retour.

Maintenant que l'intégration des nouvelles routines de protection dans le *benchmark* a été expliquée, intéressons-nous à la méthode de simulation des fautes que nous intégrons cette fois-ci dans l'émulateur.

## B.2. Simulation de fautes dans l'émulateur – Les différents scénarii d'apparition de fautes

Comme nous l'avons évoqué précédemment, nous sommes dans l'hypothèse où il y a un mécanisme de détection matériel qui déclenche une interruption spécifique au processeur, à chaque fois où il détecte une erreur. Nous proposons ainsi quelques scénarii d'apparition de fautes. Nous ajoutons dans l'émulateur une fonction supplémentaire qui a pour rôle de choisir les instants de ces interruptions. Nous avons proposé de tester quelques diverses possibilités, à savoir des interruptions périodiques ou aléatoires pendant toute la durée de l'exécution du programme, des interruptions périodiques qui apparaissent pendant un intervalle bien déterminée (salve d'erreurs) ou aussi des interruptions aléatoires qui apparaissent pendant un intervalle choisi aléatoirement. Ainsi, quatre différents scénarii d'apparition de fautes sont développés (Figure 3.4) pour l'analyse et l'évaluation des performances :

- Injection périodique des erreurs pendant la durée du programme : une erreur toutes les N instructions (Scénario 1). Nous pouvons varier N ce qui nous donne plusieurs scénarii.
- Injection aléatoire des erreurs pendant la durée du programme : les erreurs ne sont plus périodiques (Scénario 2).
- Injection par salve :
    - Choix d'intervalles d'apparition d'erreurs et injection périodique (Scénario 3)
    - Intervalle d'apparition d'erreurs choisi aléatoirement et injection aléatoire (Scénario 4)

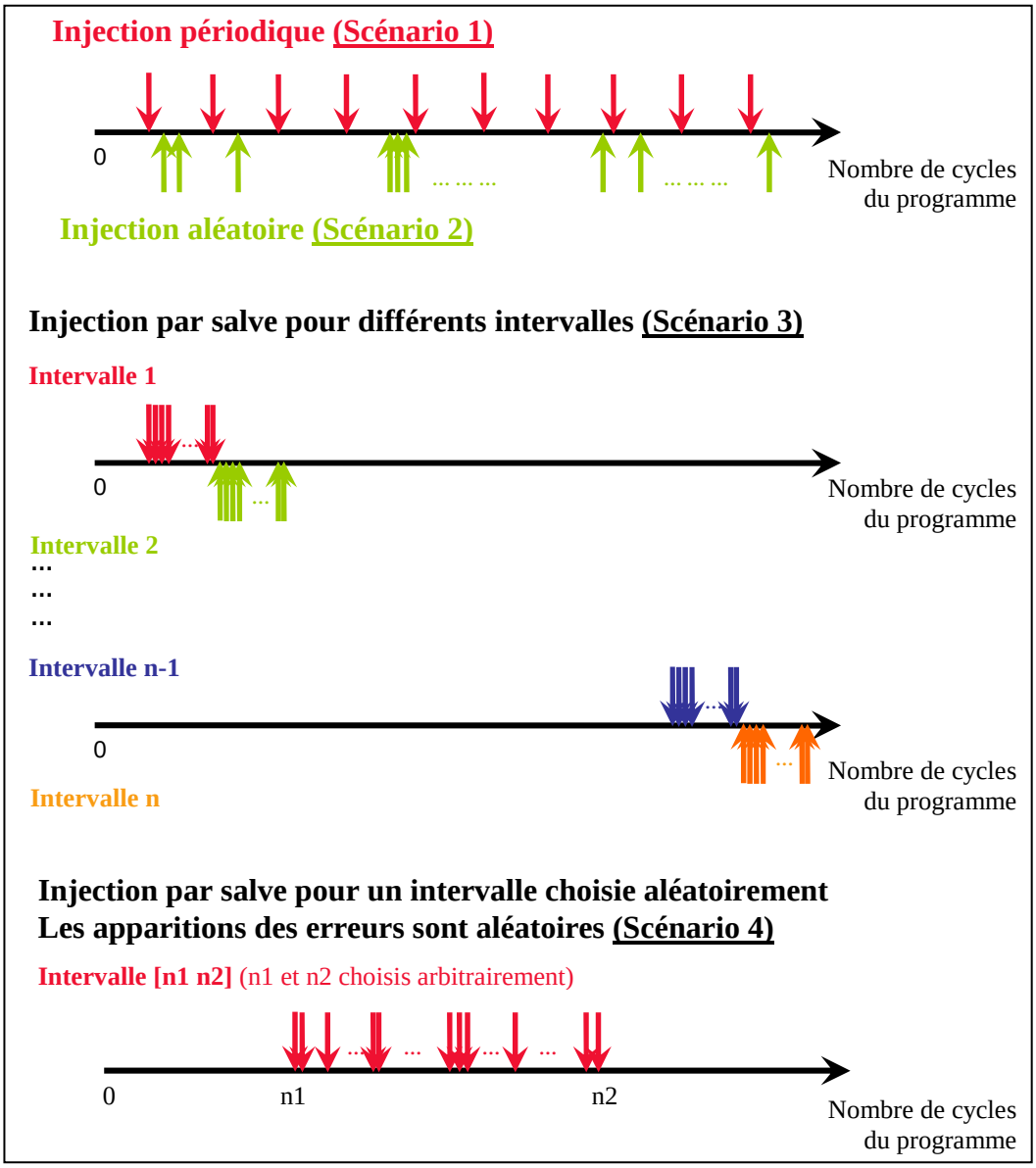


**Figure 3.4 : Les différents scénarii d'apparition d'erreurs implantés dans l'émulateur**

Concernant l'injection d'erreurs par salve, dans certains cas de figures, en particulier la salve avec erreurs nombreuses, avec l'utilisation de cette technique de protection basée sur le recouvrement logiciel, le surcoût temporel risque de décroître bien que le nombre d'erreurs augmente. En effet, lors d'une salve d'erreurs nombreuses, dès que la première routine de correction commence à s'exécuter suite à une première interruption, une seconde interruption risque de se déclencher. De ce fait, la routine de protection se ré exécute une seconde fois sans pour autant la première ait fini. Donc, nous pouvons dire que lors de quarante erreurs successives par exemple, ce n'est pas forcément les quarante routines qui s'exécutent, c'est possible que ça soit uniquement la dernière selon la périodicité entre les erreurs. Schématisons ceci par les différents chronogrammes illustrés dans la figure 3.5. Supposons que la routine de

protection (retour au dernier état sûr) dure quatre unités de temps, et que l'application sans cette routine de protection dure 20 unités de temps.

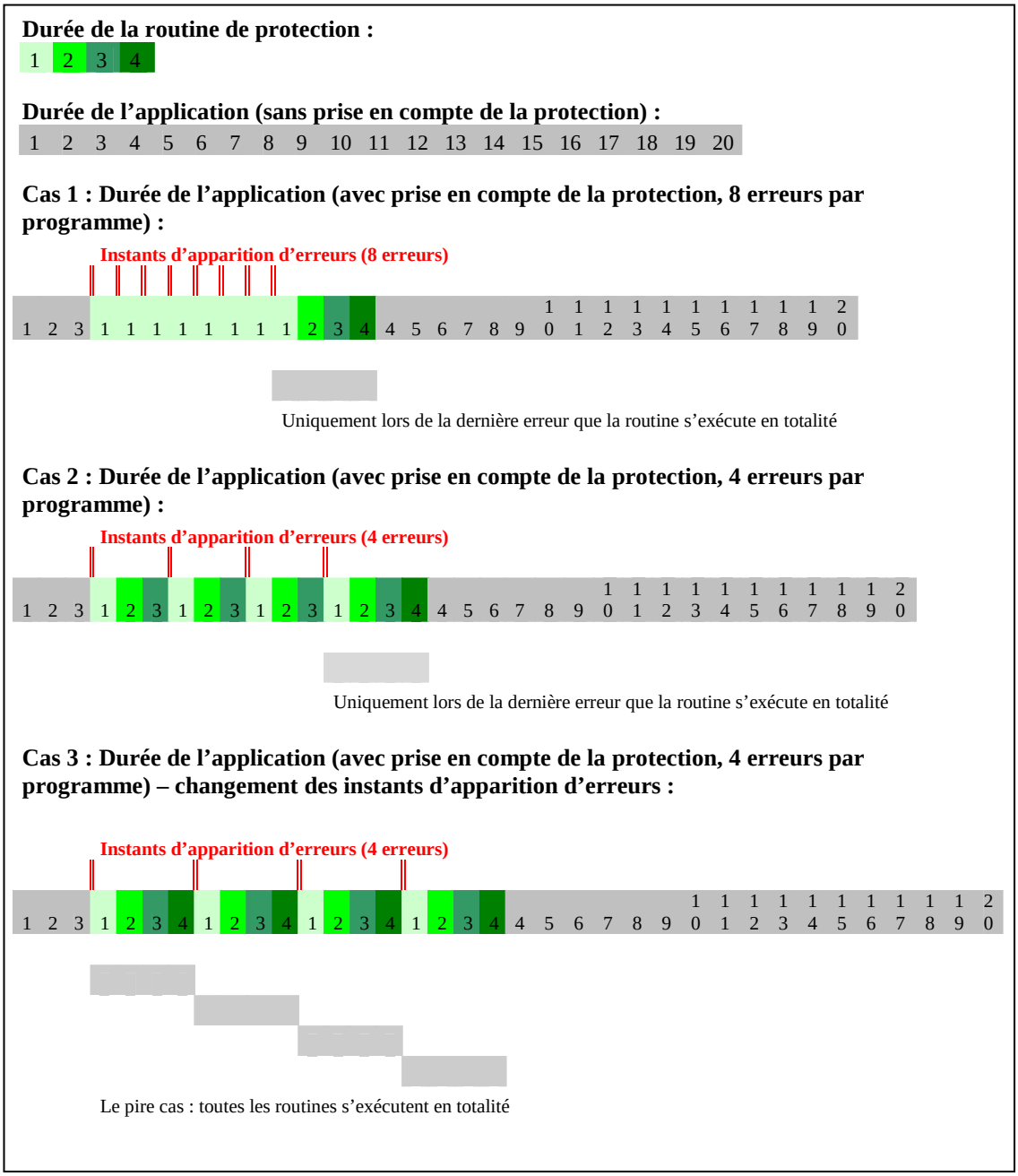


**Figure 3.5 : Différence entre les surcoûts temporels selon les instants d'apparition d'erreurs**

Nous remarquons bien que l'application tolérante aux fautes, avec une injection de huit erreurs par programme (cas 1) dure 31 unités de temps par rapport à une durée de 20 unités de temps pour un programme sans apparition d'erreurs. Celle dont nous y avons injecté quatre erreurs (cas 2) dure 33 unités de temps, plus que celle avec huit erreurs. Le surcoût temporel est plus important pourtant le nombre d'erreurs est plus faible. Ce qui explique ce que nous venons de dire ci-dessus. Nous poussons l'exemple encore plus loin en jouant sur les instants

d'apparition d'erreurs, nous arrivons même à atteindre 36 unités de temps. Nous allons vérifier dans la suite si nous aboutissons au même résultat lorsque nous injectons les scénarii 3 et 4 d'apparition d'erreurs dans l'émulateur.

Les routines de protection étant intégrées dans le *benchmark*, les simulations d'erreurs étant implantées dans l'émulateur, les différents scénarii d'apparition d'erreurs étant définis, nous nous proposons maintenant d'étudier les capacités de correction de cette technique de protection ainsi que ses surcoûts temporels selon les différents scénarii d'apparition d'erreurs.

## B.3. Capacités de correction et surcoûts temporels de cette technique de protection

Intéressons-nous aux conséquences temporelles de cette technique. Le tableau suivant résume, pour les deux stratégies mémoire, les surcoûts temporels en citant en premier lieu le nombre de cycles dans le cas d'un *benchmark* sans protection, en second lieu, le cas d'un *benchmark* avec une protection périodique et en dernier lieu, le cas avec une protection avec une erreur toutes les N  Instructions.

| | **Nombre de cycles** | | **Surcoût de l'architecture 2 bus (%)** | **Surcoût de la protection (%)** | |
|---|---|---|---|---|---|
| | **3 bus** | **2 bus** | | **3 bus** | **2 bus** |
| **1er cas : résultats sans protection** | 1833 | 2333 | 27% | | |
| **2ème cas : résultats avec sauvegarde périodique du contexte** | 2367 | 2987 | 26% | 29% | 28% |
| **3ème cas : résultats avec sauvegarde périodique du contexte et protection à chaque erreur (une erreur / 250 instructions)** | 2694 | 3455 | 28% | 47% | 48% |

**Tableau 3.3 : Surcoût de la protection**

Nous constatons que nous atteignons un surcoût de 48% avec au maximum dix erreurs corrigeables. Ce qui nous impose par exemple d'avoir recours à d'autres techniques moins gourmandes en temps au cas où l'application comporte des contraintes temporelles (par exemple, pas plus que 30% de surcoût en temps).

Ceci est le cas pour une erreur toutes les 250 instructions, correspondant à dix erreurs par programme. Ce nombre de dix est trouvé à travers l'ajout d'un compteur de nombre d'erreurs dans l'émulateur. Le nombre d'erreurs a varié d'une erreur toutes les 1000 instructions jusqu'au taux d'erreurs maximum corrigeable. Nous obtenons ainsi l'histogramme suivant (Figure 3.6). Il montre l'évolution des surcoûts temporels en fonction du nombre d'erreurs.

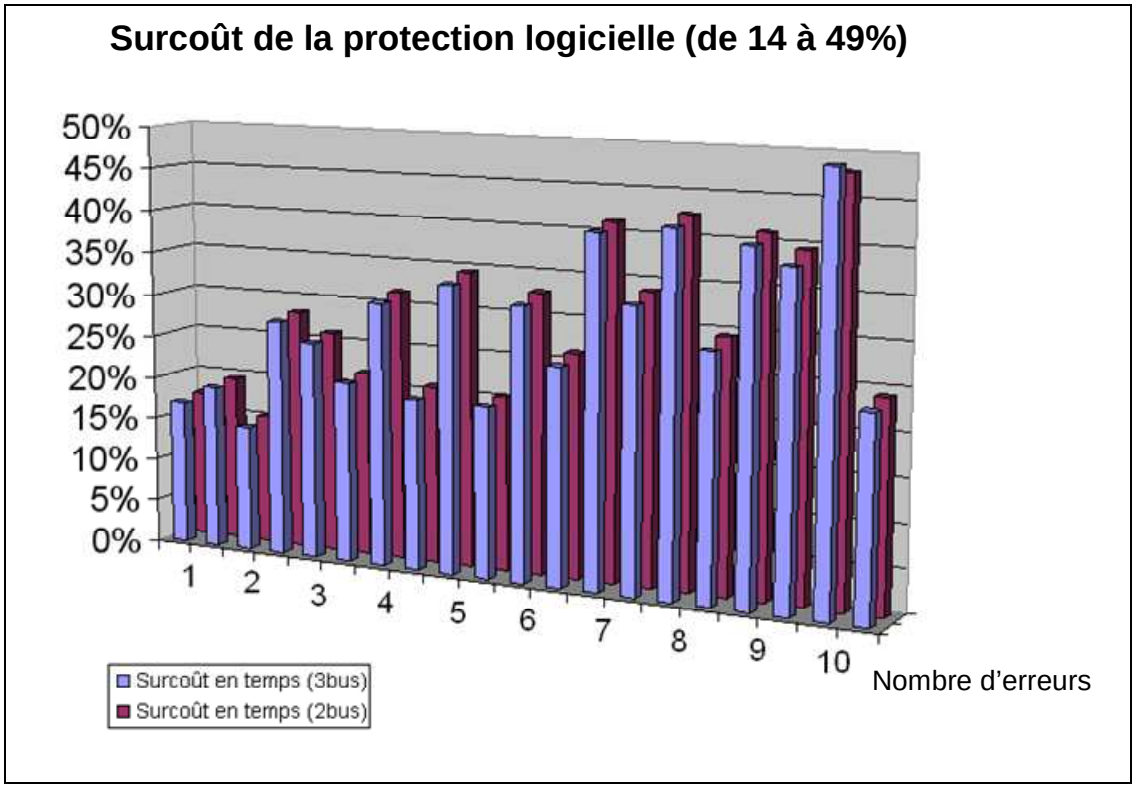


**Figure 3.6 : Surcoûts temporels de la correction pour une apparition d'erreurs périodique (scénario 1)**

Nous remarquons que le surcoût temporel varie de 14 à 49% en fonction du nombre d'erreurs. Par contre, le plus intéressant à voir c'est qu'on ne peut pas corriger plus de dix erreurs se produisant pendant la durée d'exécution d'un programme. Le taux maximum d'erreurs corrigeables est d'une erreur toutes les 207 instructions ; au-delà le programme s'occupe plutôt à faire une sauvegarde et un retour au dernier état sûr sans continuer les traitements de l'application. Dans un objectif de remplir un cahier de charges, nous pouvons conclure que si la contrainte temps réel de l'application exige à ne pas dépasser un surcoût de 35% par exemple, cette technique ne peut corriger que 6 erreurs. Si par ailleurs nous sommes dans un environnement où le taux d'erreurs est quantifié, disons de l'ordre de 11 erreurs/ms, et en respectant la même contrainte temps-réel, l'application doit tourner en 0,5ms. Autrement, les erreurs ne sont plus corrigeables. En effet, la technique de protection permet de corriger six erreurs par programme. Le programme dure 0,5ms donc 6 erreurs/0,5ms sont corrigeables. Par conséquent, 12 erreurs/ms sont corrigeables ce qui est supérieur au taux d'erreur de l'environnement ou se situe le processeur (11 erreurs/ms). Si par contre l'application tourne en 0,6 ms, 6 erreurs/0,6ms, autrement dit 10 erreurs/ms, sont corrigeables, ce qui n'atteint pas le seuil.

Ce que nous essayons de montrer ici c'est que connaissant les contraintes temps-réel, en connaissant également l'environnement qui entoure le système avec un taux d'erreur défini,

nous réussissons à trouver les capacités de correction et le seuil à partir duquel le processeur ne remplit plus son service.

En changeant la méthode d'injection d'erreurs en la rendant aléatoire (figure 3.7), nous arrivons à corriger 73 erreurs avec 200% de surcoût, ceci sans la sauvegarde des données triées. C'est à dire que la fréquence est divisée par trois. En prenant en compte dans la sauvegarde du contexte les données traitées (à savoir, les dix données à trier), nous arrivons à corriger 249 erreurs. Toutefois, le surcoût temporel pour ce maximum d'erreurs corrigeables est de 1000%. Par contre, si nous avons par exemple une contrainte temps-réel qui nous exige de ne pas dépasser un surcoût de 130%, dans un tel cas, le nombre d'erreurs corrigeable est de neuf erreurs par programme. Au-delà, nous perdons énormément en performances temporelles.

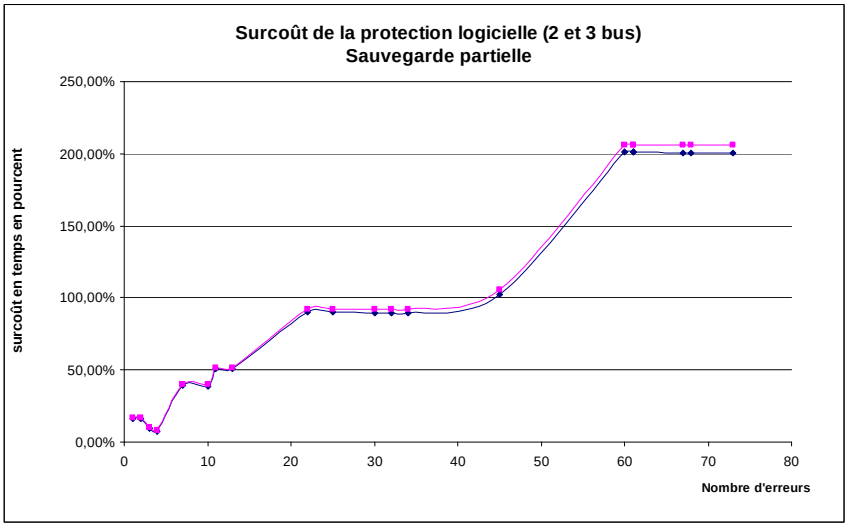


**Figure 3.7 : Surcoût de la protection logicielle (scénario 2)**

Cette fois-ci, nous nous proposons d'injecter des erreurs de façon fréquente (scénario 3) ou aléatoire (scénario 4) pendant un intervalle déterminé. Cet intervalle peut également être aléatoire. Pour le scénario 3, dans la mesure où l'exécution normale du programme de tri sans protection dure 1833 cycles (en 3 bus), nous avons alors choisi d'injecter des erreurs pendant des intervalles de 200 cycles et ce en changeant l'instant de début de l'intervalle de 0 à 1600. Nous relevons les résultats selon la fréquence des erreurs (chaque 5 cycles, 10 cycles ou 20 cycles), qui sont illustrés par la Figure 3.7. Évidemment, d'autres tests peuvent être effectués en changeant l'instant d'apparition de la première erreur ainsi que la durée de la salve.

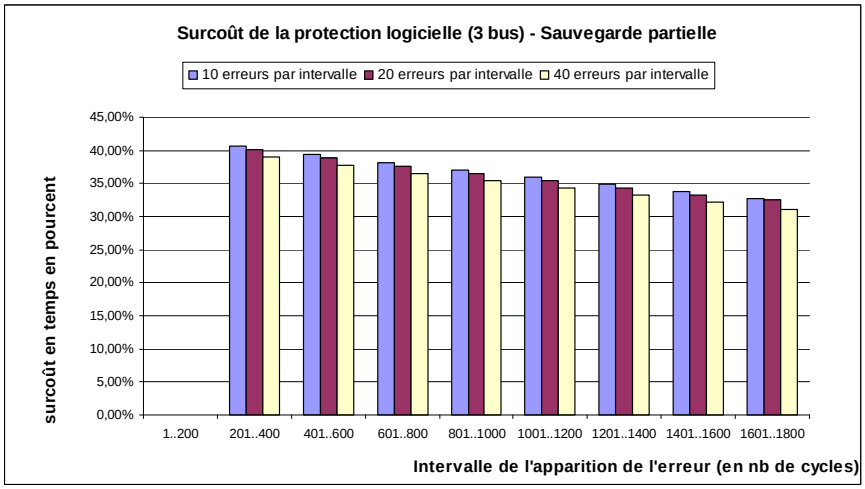


**Figure 3.8 : Surcoût de la protection logicielle (scénario 3)**

Ce qui est intéressant à noter, c'est que pour des erreurs qui apparaissent dès le départ, la technique ne réussit pas à les corriger. Ceci est dû au fait que la première sauvegarde du contexte ne se fait qu'après 32 cycles ne se sont écoulés. Donc, une erreur qui apparaît avant 32 cycles doit être corrigeable autrement. Le plus judicieux est, par exemple, d'ajouter une fonction qui fait un retour à l'état initial. La seconde constatation concerne la différence des surcoûts temporels pour un intervalle donné. En effet, l'exécution du programme avec une correction de dix erreurs dure un peu plus que celui avec une correction de 40 erreurs (une durée supplémentaire de 1 à 2% pour tous les intervalles étudiés et ceci pour les deux cas sauvegarde partielle et complète). Ceci paraît a priori illogique puisque théoriquement 40 erreurs à corriger, c'est-à-dire l'exécution de 40 fois la fonction de retour au dernier état sûr, demande plus de cycles que son exécution pendant 10 fois. Ce qui se passe c'est qu'aucune de ces 40 retours au dernier état sûr n'arrive à sa fin avant qu'une autre demande de ré-exécution de cette fonction réapparaisse. Ceci justifie ce que nous avons précédemment exposé dans la partie B.2. La protection d'une salve d'erreurs nombreuses et successives correspond quasiment à la protection d'une seule (la dernière) erreur.

Cette fois-ci, nous nous proposons d'injecter des erreurs d'une manière aléatoire pendant un intervalle arbitraire (scénario 4). Cet intervalle est le suivant : [722 ; 1540] cycles. La borne supérieure ne dépasse pas les 1833 cycles dans la mesure où l'exécution normale du programme de tri sans protection dure 1833 cycles (en 3 bus).

Nous avons alors choisi d'injecter des erreurs aléatoirement pendant cet intervalle et ce, d'un point de vue algorithmique, de manière à ce que les nombres tirés aléatoirement soient inférieurs à un certain seuil. La variation de ce seuil nous permet d'avoir différents nombres

d'erreur possibles. Nous relevons ainsi les résultats des surcoûts selon ce nombre d'erreurs, autrement dit selon la durée de la salve d'erreurs. Ces résultats sont illustrés par la Figure 3.9.

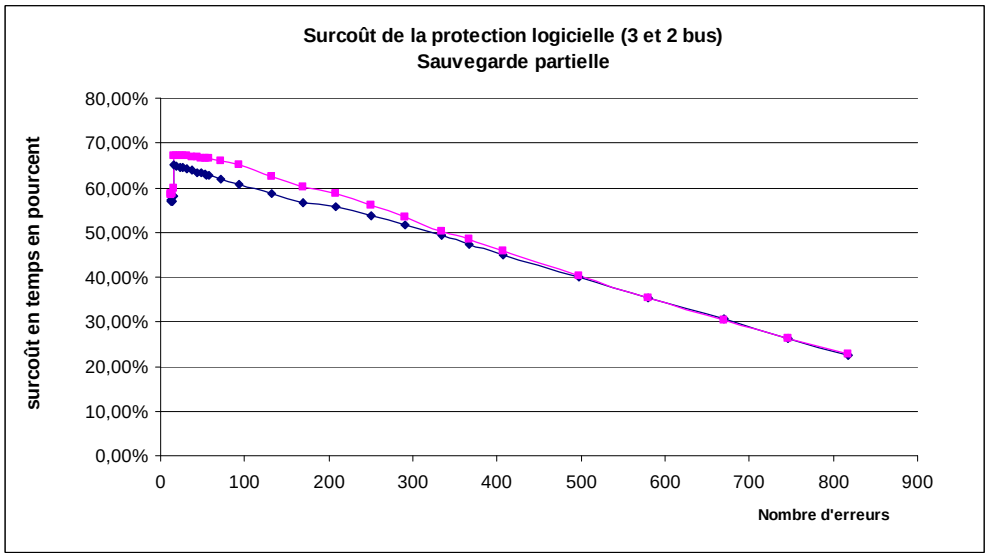


**Figure 3.9 : Surcoût de la protection logicielle (scénario 4)**

Nous remarquons que le surcoût est inférieur à 70% pour la sauvegarde partielle et qu'il est inférieur à 103% pour la sauvegarde complète. Ce qui est aussi intéressant à noter, c'est que si les fréquences des erreurs sont importantes, les routines de protection envers ces erreurs ne s'exécutent pas en totalité. Alors qu'avec des fréquences d'erreurs faibles, les routines de protection de ces dernières sont exécutées en totalité, d'où l'explication de la décroissance du surcoût, introduite en B.2 de ce chapitre (commentaires qui suivent la Figure 3.5).

Nous pouvons ainsi faire une synthèse de ces résultats dans ce tableau :

| Injection d'erreurs périodique | Injection d'erreurs aléatoire | Injection d'erreurs par salve |
|---|---|---|
| • La non croissance du surcoût est faussée par le phénomène cyclique des erreurs | • Le surcoût en temps commence à être pénalisant et plus gênant que des erreurs par salve<br>• L'intention de trouver un compromis matériel/logiciel est confirmée à cause du surcoût excessif à partir d'un certain nombre d'erreurs | • Les erreurs par salve ne sont pas aussi gênantes que celles qui apparaissent d'une manière aléatoire<br>• Le surcoût d'un nombre important d'erreurs qui apparaît pendant une courte durée est moins coûteux que lorsque ce même nombre d'erreurs apparaît aléatoirement |

**Tableau 3.4 : Synthèse des résultats selon les scénarii d'injection d'erreurs**

Ces résultats peuvent aussi être présentés d'une autre manière en relevant le taux d'erreurs maximal que cette technique peut corriger (T.e.m), le surcoût temporel correspondant à ce taux (S.t) et le surcoût temporel maximal correspondant à ce type

d'injection (S.t.max) selon le scénario d'apparition d'erreurs : périodique (I1), aléatoire (I2) ou par salve (I3). L'exemple traité est le même programme de tri de dix variables. Nous rappelons que son exécution dure 1833 cycles d'instructions.

| | **I1** | **I2** | **I3 (intervalle choisi)** | **I4 (intervalle arbitraire)** |
|---|---|---|---|---|
| **T.e.m** | 10 erreurs par programme | 73 erreurs par programme | 40 erreurs par 200 cycles | 818 erreurs par 818 cycles |
| **S.t** | 23,46% | 200,44% | 39,12% | 22,48% |
| **S.t.max** | 49,10% | 200,76% | 40,70% | 65,25% |

**Tableau 3.5 : Taux et surcoûts de la protection logicielle**

Ce que nous constatons, c'est qu'au pire des cas, la correction est possible pour dix erreurs par programme ce qui correspond à 1 erreur toutes les 207 instructions. Ce qui nous permet de conclure que cette protection logicielle est plus qu'avantageuse, et ça ne nécessite pas d'autre protection matérielle sauf si on est au delà de ce taux. Quant aux surcoûts temporels, nous atteindrons 200% de surcoût pour 73 erreurs corrigées, c'est à dire que la fréquence est divisée par trois. Pour ce type d'injection, nous atteindrons 50,57 % pour 13 erreurs corrigées. Par contre, si l'application est située dans un environnement plus perturbé qui nous donne un taux d'erreur supérieur à une erreur toutes les 207 instructions, ceci nous impose d'avoir recours à d'autres techniques de protection matérielles.

# C. Conclusions

Nous venons de quantifier les surcoûts temporels de la technique de correction que nous avons développée et intégrée dans le *benchmark*. Nous avons conclu que si nous disposons de certaines spécifications du cahier de charge, tels que les contraintes temps-réel et le taux d'erreur de l'environnement ou se situe le processeur, nous pouvons justifier l'efficacité de notre technique de protection et nous pouvons donner jusqu'à quelle limite le processeur ne remplit plus son service. Notons que cette technique de protection est indépendante du fichier assembleur et donc indépendante de l'application. C'est un bloc logiciel indépendant, il suffit juste de modifier les adresses de retour. Cette technique de correction est indépendante du moyen de détection. Il suffit juste que cette détection informe le processeur via une interruption externe. Cette démarche s'inscrit alors dans un aspect méthodologique. Nous avons également défini et justifié le choix de la stratégie trois bus en se basant également sur l'ensemble émulateur/*benchmark*.

Nous nous proposons dans le chapitre suivant de converger nos travaux avec ceux de l'équipe du CRAN de Nancy/A3SI-ENSAM de Metz dans le cadre de la collaboration sollicitée par le projet CIM'Tronic. Il s'agit de valider la technique de protection logicielle par le biais d'une approche de type flux informationnel.

# Chapitre 4
# Application de l’approche du flux informationnel sur le modèle du processeur

Un des objectifs importants et fixés par le projet CIM’tronic consiste à intégrer les travaux de divers partenaires en un travail commun. C’est dans ce contexte que ce chapitre est dédié à la validation de l’approche que nous avons proposée et implantée dans les deux chapitres précédents. Nous allons ainsi exploiter l’approche du flux informationnel fournie par l’équipe de recherche du CRAN de Nancy/A3SI-ENSAM de Metz en se basant sur les chemins de données des instructions définies précédemment ainsi que leurs étapes d’exécution.

## A. Présentation de la démarche d’évaluation fiabiliste

Dans ce chapitre, nous évaluons certains paramètres de sûreté de fonctionnement d’une application embarquée dans un processeur à piles. En effet, en s’inspirant de la norme [IEC99], [BELH08] a défini six modes de fonctionnement grâce à une analyse dysfonctionnelle. Ces modes sont :

- Mode 1: mesure correcte, pas de défaillance détectée.
- Mode 2: mesure incorrecte, défaillance détectée mais tolérée (recherche de disponibilité).

— Mode 3: mesure incorrecte, défaillance détectée mais non tolérée (recherche de sécurité).
— Mode 4 : mesure incorrecte, défaillance non détectée (mode dangereux).
— Mode 5: mesure correcte et défaillance détectée (arrêt intempestif).
— Mode 6: absence de mesure (arrêt du système).

Ces paramètres de sûreté de fonctionnement que nous nous proposons d'établir sont les probabilités de présence dans un de ces six modes de fonctionnement. Pour ce faire, nous utilisons deux outils. Le premier est l'émulateur du processeur à piles, défini dans le deuxième chapitre, qui permet l'évaluation d'une technique de tolérance aux fautes. Le second est l'approche du flux informationnel qui évalue la probabilité de défaillance de chaque instruction et par conséquent des routines du programme. L'objectif final est d'estimer la probabilité de défaillance de l'application globale. Pour cela, nous nous proposons en premier lieu d'appliquer l'approche du flux informationnel sur une instruction du jeu d'instructions du processeur à piles à partir de son schéma d'exécution et d'estimer sa probabilité. En second lieu, nous généralisons cette évaluation sur l'ensemble des instructions pour aboutir à la probabilité de tout le programme embarqué décrivant l'algorithme de tri, défini dans le troisième chapitre. En dernier lieu, nous intégrerons la routine de protection dans la modélisation et le calcul pour voir son influence sur les diverses probabilités.

Pour appliquer l'approche du flux informationnel, nous considérons l'architecture du processeur à piles comme cas d'étude. Nous rappelons que la première pile est utilisée pour le traitement de données. Quant à la seconde, elle est utilisée pour les retours d'adresses des fonctions et des copies temporaires de données. Les éléments des piles sont gérés dans une mémoire externe au cœur du processeur pour éviter la restriction au niveau de la profondeur des piles. Ces dernières sont adressées par des pointeurs internes : pointeur de pile de données (DSP : *Data Stack Pointer*) et pointeur de pile de retour (RSP : *Return Stack Pointer*). Le chemin de données et le schéma d'exécution de chaque instruction sont ainsi établis afin d'aborder ensuite la modélisation selon l'approche du flux informationnel.

La première étape de la modélisation selon l'approche du flux informationnel consiste à réaliser le modèle de haut niveau établi par des entités sous-fonctionnelles qui identifie les ressources matérielles utilisées. Ce modèle de haut-niveau caractérise l'échange d'information qui s'effectue lors de l'exécution de l'instruction (flux de données, évènements de contrôle, stockage d'information…etc.) entre les différentes entités sous-fonctionnelles. La seconde étape consiste à établir le modèle de bas niveau. Il s'agit de créer, pour chaque entité sous-

fonctionnelle du modèle de haut-niveau, un automate à états finis décrivant le comportement de cette entité. À partir de ce modèle de bas niveau, nous déterminons les différents scénarii de défaillances possibles. Ces scénarii sont ensuite regroupés par liste selon les six modes de fonctionnement cités ci-dessus. Ces listes sont ensuite transformées en arbres de défaillance. La probabilité de présence d'une instruction dans un de ces six modes de fonctionnement est alors calculée à partir de probabilités de défaillance de l'élément au sommet des divers arbres de défaillance. L'application de cette approche du flux informationnel sur une instruction du jeu d'instructions du processeur est résumée dans la figure 4.1 suivante.

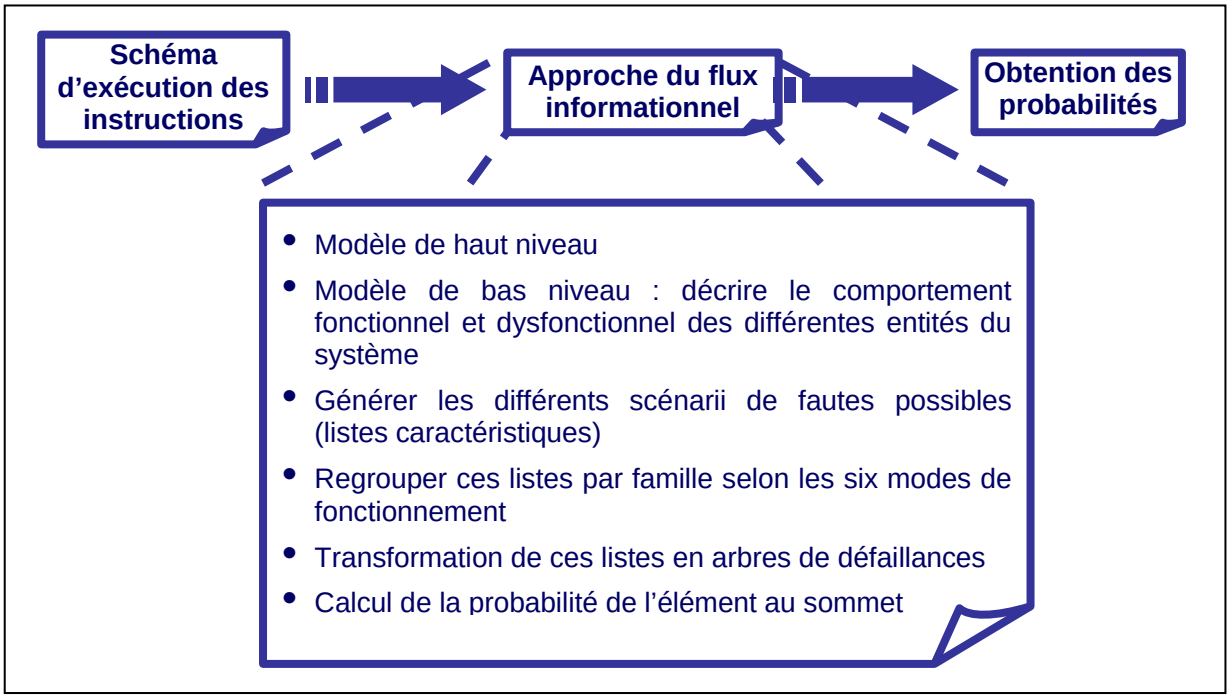


**Figure 4.1 : Stratégie globale de l'évaluation de la probabilité de défaillance d'une instruction**

Dans la mesure où notre objectif est d'évaluer la probabilité de tout le programme de tri incluant la méthode de protection, l'évaluation faite sur une seule instruction est généralisée sur l'ensemble des instructions. Elle sert ainsi à l'évaluation des probabilités des diverses fonctions composant le programme de tri. Nous rappelons l'algorithme de ce programme dans la figure qui suit.

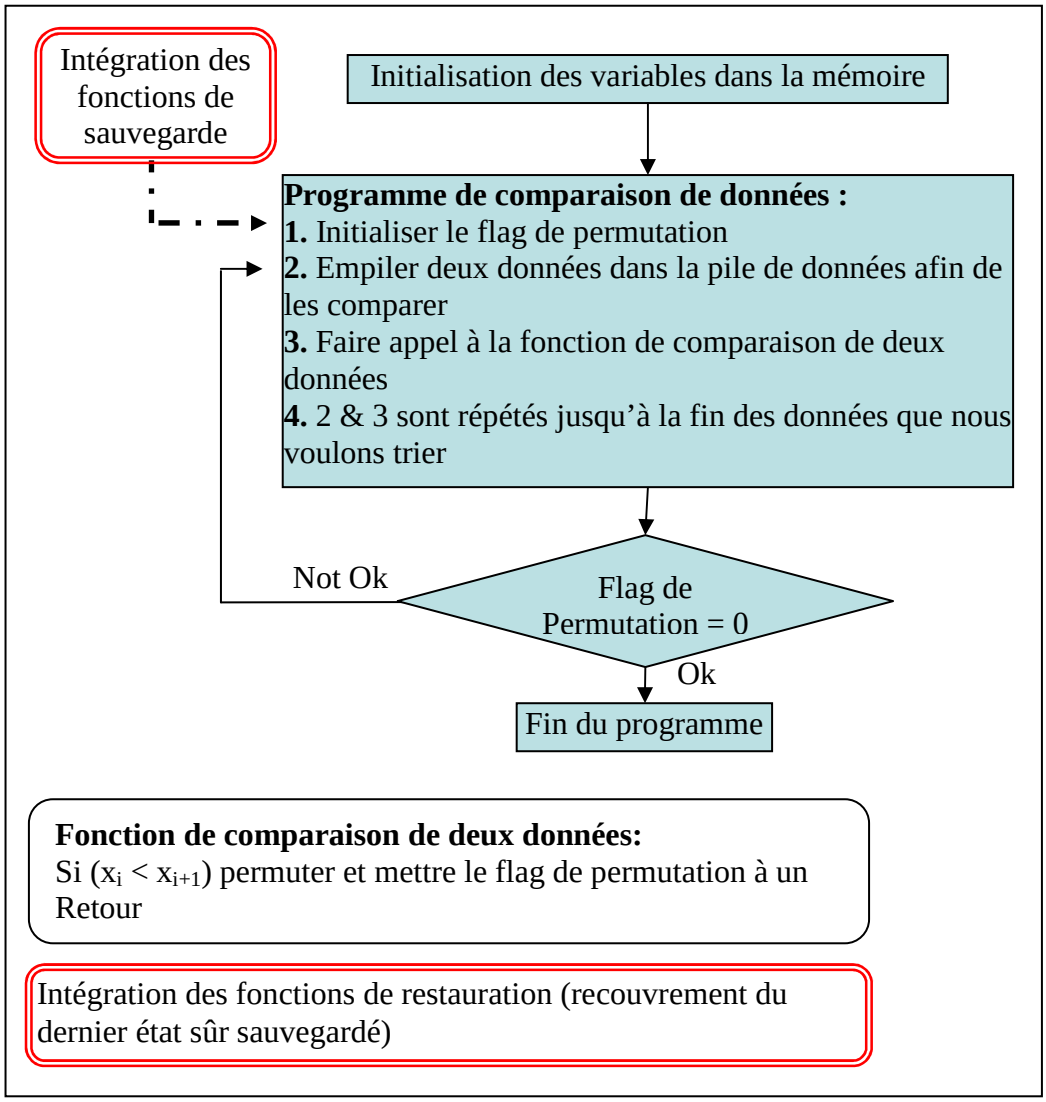


**Figure 4.2 : Algorithme de tri à bulles avec prise en compte de la tolérance aux fautes**

Parmi ces fonctions, nous trouvons la fonction d'initialisation, la fonction de comparaison de dix données, la fonction de test du flag de permutation, la fonction de comparaison de deux données, la fonction de sauvegarde et la fonction de restauration. Ce qui permettra d'évaluer la probabilité de présence dans un des six modes de fonctionnement du programme complet. Cette démarche est résumée dans la figure suivante.

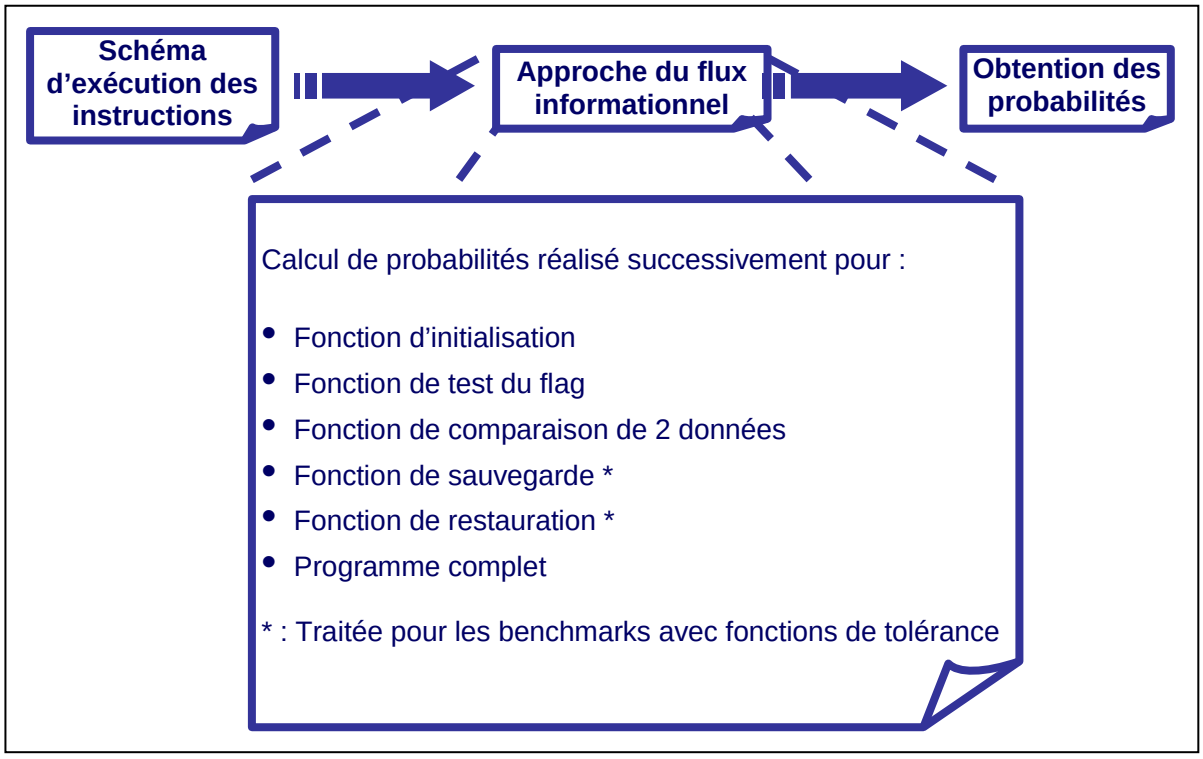


**Figure 4.3 : Stratégie globale de l'évaluation de la probabilité de défaillance du programme complet**

Les détails de la démarche globale de l'évaluation de la sûreté, incluant l'émulateur et l'approche du flux informationnel, sont présentés dans la Figure 4.4. À partir des spécifications et des besoins, une architecture matérielle est proposée et son jeu d'instruction est conçu. Chaque instruction est modélisée par son schéma d'exécution et son chemin de données. Le flux d'information est ainsi établi afin de faciliter la transition vers le modèle de haut niveau et par la suite le modèle de bas niveau. Comme nous l'avons précédemment détaillé, l'émulateur sert, entre autres, à simuler l'injection de fautes et de mesurer les conséquences temporelles de la technique de protection implantée dans le *benchmark*, désignant une application mécatronique. De l'autre côté, à l'aide des modèles de bas niveau, nous calculons les probabilités d'existence de l'application dans un des six modes de fonctionnement. Pour cela, nous détaillons tout d'abord les chemins de données et les schémas d'exécution de certaines instructions, ce qui sera l'objet de la partie suivante. Évidement, à partir d'une analyse globale de sûreté, des éventuelles améliorations sur l'architecture matérielle (jeu d'instruction) – logicielle (émulateur/*benchmark*) peuvent s'avérer nécessaires.

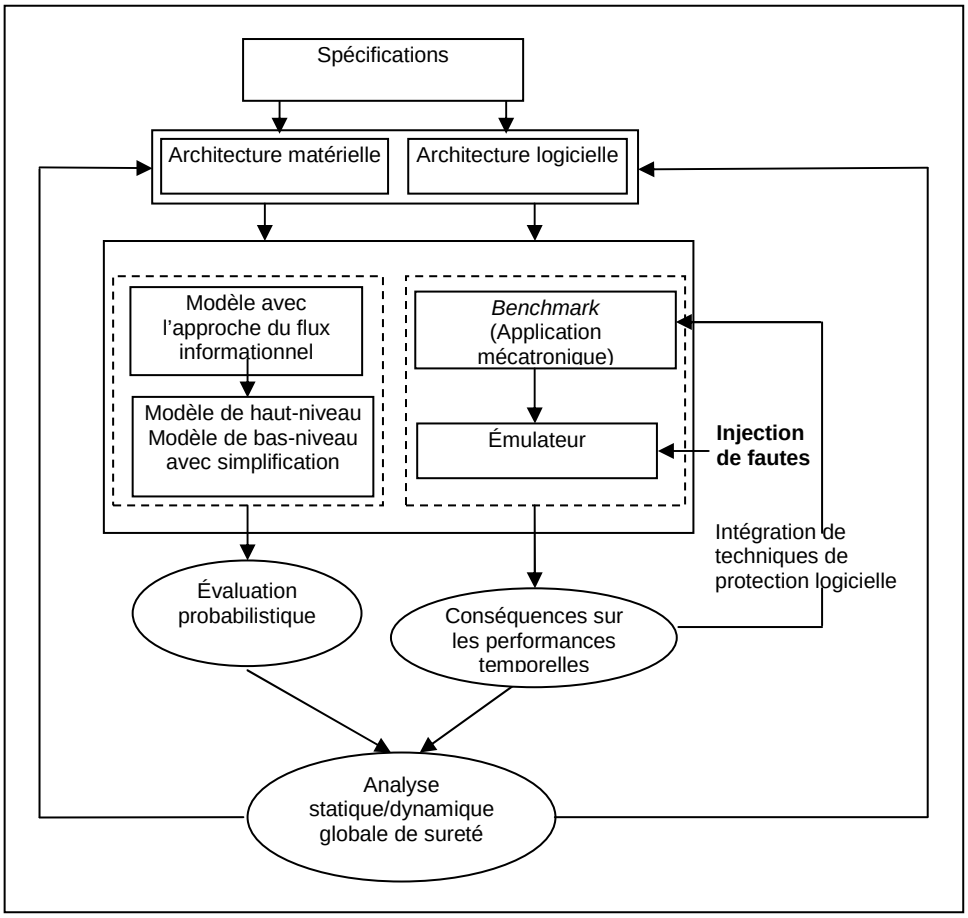


**Figure 4.4 : Démarche globale de l'évaluation de la sûreté**

# B. Chemins de données et schémas d'exécution de certaines instructions

Évidemment, nous ne donnerons pas dans ce mémoire les chemins de données de toutes les 38 instructions. Nous nous contentons des exemples précis : les instructions arithmétiques et logiques, qui traitent deux opérandes, dans la mesure où ils sont très utilisés, l'instruction de duplication du sommet de la pile DUP, exploitant un seul opérande et s'exécutant en un seul cycle d'horloge et enfin l'instruction de stockage STORE qui, quant à elle, nécessite deux cycles d'horloge pour être exécutée.

Commençons par les instructions arithmétiques et logiques, comme c'est illustré dans la Figure 4.5.

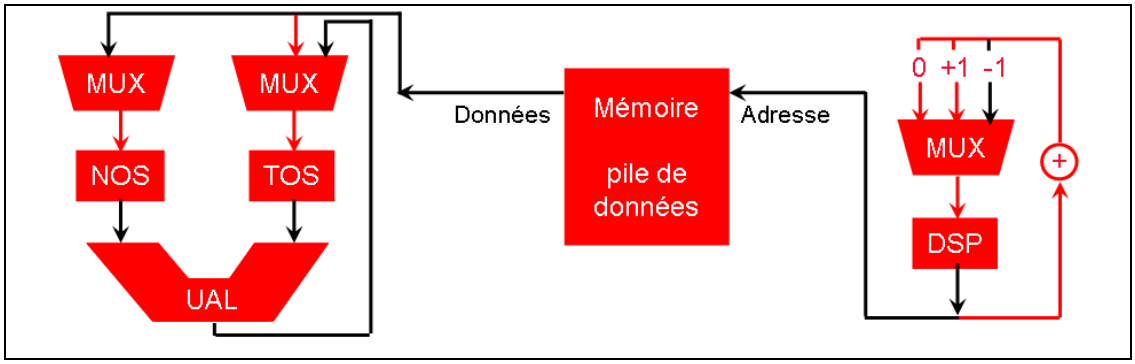


**Figure 4.5 : Chemin de données d'instructions arithmétiques et logiques manipulant TOS et NOS**

Pour comprendre au mieux l'enchaînement des étapes, détaillons les opérations une par une :

- TOS ← NOS (+/- AND OR …) TOS
- NOS ← 3ème élément de la pile de données (c'est l'élément mémoire adressé par le pointeur de pile de données DSP) *« 3ème élément de la pile de données devient alors NOS »*
- DSP ← DSP - 1 « décrémentation du DSP parce qu'un élément vient d'être dépilé et placé au NOS »

Concernant les signaux de contrôle, ils doivent être de telle manière que :

- La mémoire pile de données en mode lecture (enable)
- Le multiplexeur en amont du pointeur de pile de données DSP (Mux_DSP) est en position '-1'
- Le multiplexeur en amont du sous-sommet de la pile de données DSP (Mux_NOS) est en position 'lecture à partir du TOS'

Au top d'horloge, toutes les opérations sont réalisées et l'instruction est exécutée :

- TOS ← NOS (+/- AND OR …) TOS
- NOS ← MEM[DSP] (3ème élément)

Maintenant nous abordons autres types d'instructions, celles manipulant un seul opérande, comme c'est le cas de l'instruction DUP. Elle permet la duplication du sommet de la pile TOS. Son chemin de données est illustré par la Figure 4.6 suivante.

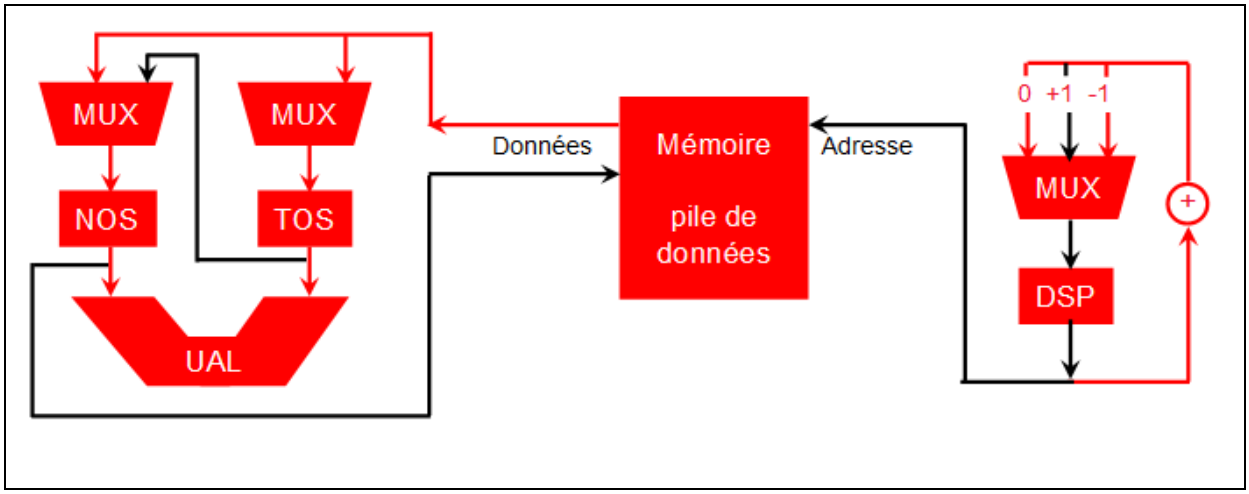


**Figure 4.6 : Chemin de données de l'instruction DUP**

L'ensemble des opérations consiste en :

- DSP ← DSP +1
- 3ème élément (MEM[nouveau DSP]) ← NOS
- NOS ← TOS

Concernant les signaux de contrôle, ils doivent être de telle manière que :

- La mémoire pile de données en mode écriture 'enable'
- Le multiplexeur en amont du pointeur de pile de données DSP (Mux_DSP) est en position '+1'
- Le multiplexeur en amont du sous-sommet de la pile de données (Mux_NOS) est en position 'lecture à partir du TOS'

Au top d'horloge, toutes les opérations sont réalisées et l'instruction est exécutée :

- MEM[DSP] (3ème Elt) ← NOS
- NOS ← TOS

Enfin, nous allons voir un autre type d'instruction nécessitant cette fois-ci deux cycles d'horloge, à savoir l'instruction STORE. Schématisons tout d'abord son chemin de données dans la Figure 4.7.

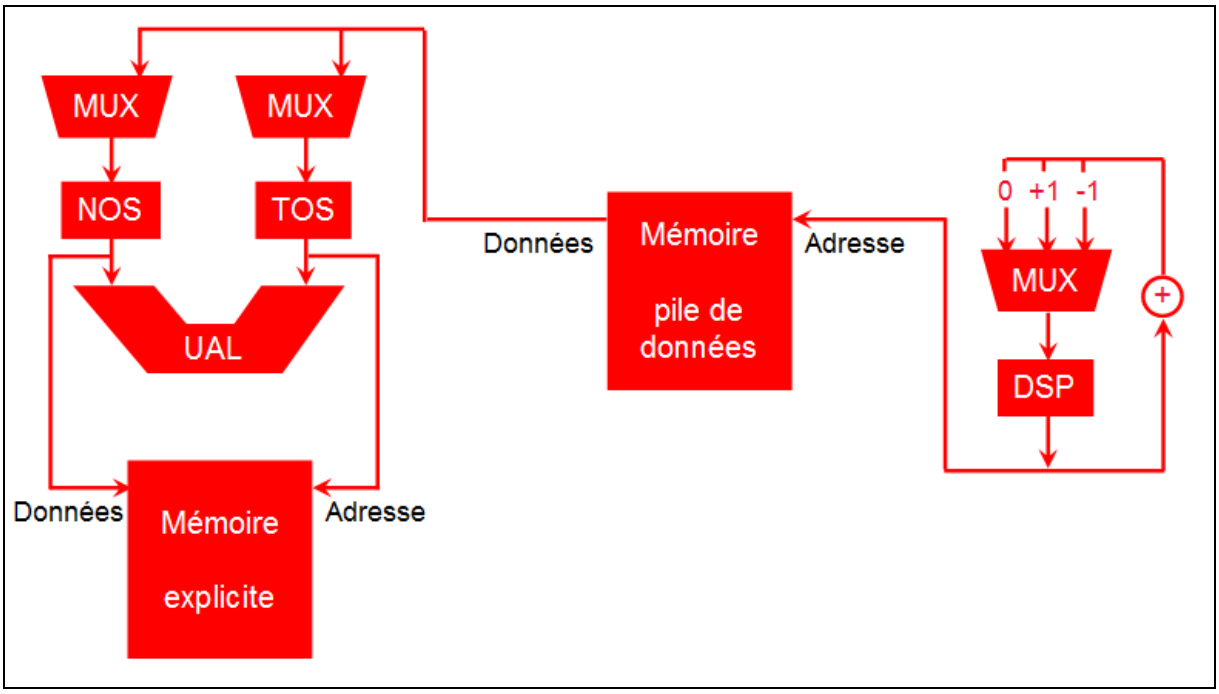


**Figure 4.7 : Chemin de données de l'instruction STORE**

L'ensemble des opérations consiste en :

- MEM[TOS] ← NOS (on écrit la valeur de NOS dans la zone mémoire adressée par le valeur de TOS)
- TOS ← 3ème élément
- NOS ← 4éme élément (TOS devient le 3ème élément de la pile et NOS le 4ème)
- DSP ← DSP -2 (double décrémentation du pointeur pile de données dans la mesure où deux éléments de la pile sont consommés)

Comme nous l'avions précisé, cette instruction nécessite deux cycles d'horloge. Ainsi pour les signaux de contrôle du premier cycle, ils seront de la sorte :

- La mémoire pile de données en mode lecture 'enable'
- La mémoire explicite en mode écriture 'enable'
- Le multiplexeur en amont du pointeur de pile de données DSP (Mux_DSP) est en position '-1'
- Le multiplexeur en amont du sommet de la pile de données (Mux_TOS) est en position 'lecture à partir de la mémoire pile de données'

Au top d'horloge, MEM[TOS] ← NOS et TOS ← 3ème élément

Pour les signaux de contrôle du second cycle, ils doivent être de telle manière que :

- Invalider la mémoire explicite

— Le multiplexeur en amont du pointeur de pile de données DSP (Mux_DSP) est en position ‘-1’

— Le multiplexeur en amont du sous-sommet de la pile de données (Mux_NOS) est en position ‘lecture à partir de la mémoire pile de données’

Au top d’horloge, le reste des opérations est réalisée : NOS ← 4$^{ème}$ élément pour enfin finir l’exécution de cette instruction.

Maintenant que les chemins de données et les diverses étapes permettant à l’exécution de quelques types d’instruction est défini, nous allons procéder à l’élaboration du modèle de haut niveau.

# C. Application de l’approche du flux informationnel sur une instruction

## C.1. Modèle de haut niveau de l’approche du flux informationnel

Dans le modèle de haut niveau, l’architecture opérationnelle est décrite. Le flux d’information est modélisé clairement à ce niveau. On distingue trois classes d'information [BELH08] :

— Information fonctionnelle (F-information).

— Information sur les erreurs détectées (D-information).

— Information de contrôle (C-information).

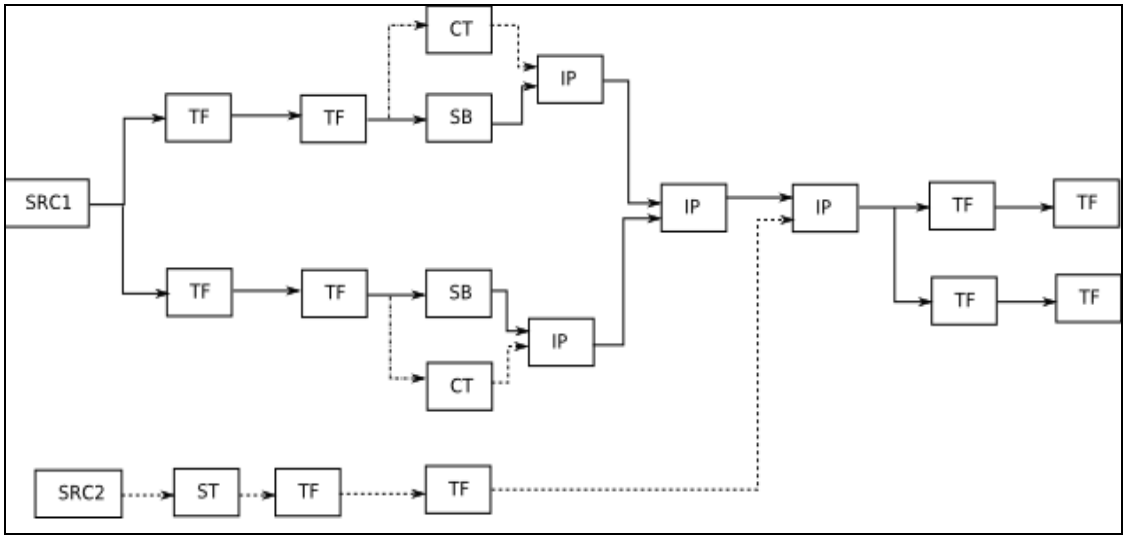


**Figure 4.8 : Exemple de modèle de haut niveau [HAMI05]**

L'information est échangée entre les entités logiques (les blocs) dans une période de scrutation bien définie. Ces blocs représentent les composants du système. Dans cette optique, nous pouvons citer plusieurs types des blocs avec des fonctions différentes [BELH08] :

- Les blocs **TF** (*Transformation*) sont utilisés pour la transformation de la nature d'information, par exemple la transformation d'un signal analogique à un signal numérique. Ils ont une entrée et une sortie. Il existe aussi un bloc TF final sans sortie.
- Les blocs **SB** (*Save Bloc*) représentent les entités fonctionnelles pour le stockage du signal.
- Les blocs **CT** (*Control*) sont utilisés pour le contrôle de flux. Ils peuvent modéliser les fonctions de vérification de la réception d'information (exemple, les chiens de garde).
- Les blocs **ST** (*Self-Test*) représentent les tests en ligne.
- Les blocs **SRC1** (*Source 1*) et **SRC2** (*Source 2*) génèrent les F- respectivement D-informations.
- Les blocs **IP** sont utilisés pour les décisions. Ils ont deux entrées et une sortie.

Maintenant que les différents constituants du modèle de haut niveau sont définis, nous donnons, à titre d'exemple, celui de l'instruction DUP.

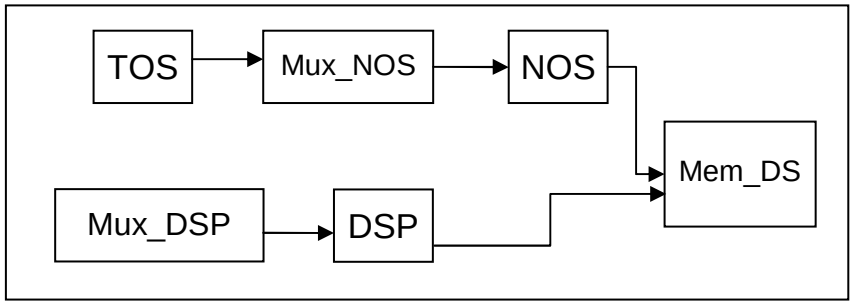


**Figure 4.9 : Modèle de haut niveau de l'instruction DUP**

L'instruction DUP copie le sommet de la pile (TOS) dans le sous-sommet de la pile (NOS) et empile ce dernier, avant qu'il soit écrasé, dans le troisième élément de la pile de données se trouvant dans la mémoire pile de données (Mem_DS). Le pointeur de pile de données (DSP) est incrémenté a travers l'entrée de sélection du multiplexeur se trouvant en amont du DSP (Mux_DSP) afin d'allouer une nouvelle zone mémoire dans Mem_DS.

À partir de ce modèle de haut niveau, nous présentons dans la suite la démarche à suivre permettant d'aboutir aux différentes probabilités de défaillances.

## C.2. Du modèle de haut niveau vers l'obtention des probabilités de défaillances

À partir de ce modèle de haut niveau, [BELH08] a établi le modèle de bas-niveau ainsi que les différents scenarii de défaillances possibles. Ces scenarii (appelés aussi listes) sont regroupés par famille suivant le mode des opérations correctes (mode 1), les modes de défaillances détectées (mode 2 et mode 3), le mode de défaillances non détectées (mode 4), le mode de défaillances tolérées (mode 2) ou non tolérées (mode 3), le mode d'arrêts intempestifs (mode 5), le mode des erreurs latentes (mode 4), le mode des pires cas (mode 4) ainsi que le mode d'absence d'information (mode 6). Ces listes sont transformées en arbre de défaillances. En effet, l'analyse en utilisant les arbres de défaillances est une technique récente et acceptée largement pour l'évaluation des probabilités et des fréquences des défaillances de système dans certaines industries [YUCH07]. La probabilité de l'élément au sommet est calculée à partir des éléments à partir des éléments de base. Un outil tel que Aralia1, donne des résultats plus précis que d'autres outils parce qu'il est 1000 fois plus rapide [GROU95]. Avec le modèle de bas niveau constitué d'automates à états finis, le taux de défaillance de la transition permettant d'aboutir à chaque état final est calculé en utilisant le processus de Markov. Les valeurs des arcs reliant deux états, représentent les taux de défaillances du flux informationnel entre ces deux états. Ces valeurs représentent aussi les éléments de la matrice de Markov. Ces valeurs calculées par le processus de Markov remplacent les différentes probabilités des éléments de base des arbres de défaillances [BELH08].

Le tableau suivant donne les probabilités d'existence dans les modes 2 et 3 pour les instructions DUP et STORE [JALL08a].

| | **Probabilité d'existence dans :** | |
|---|---|---|
| | Mode 2 : **Erreur détectée et tolérée** | Mode 3 : **Erreur détectée et non tolérée** |
| **DUP** | $72.\ 10^{-3}$ | $9.\ 10^{-5}$ |
| **STORE** | $11{,}3.\ 10^{-4}$ | $11{,}3.\ 10^{-7}$ |

**Tableau 4.1: Probabilités d'existence dans les deux modes 2 et 3 de fonctionnement**

Suivant le Tableau 4.1, d'un coté, nous constatons que la probabilité d'existence dans le mode 2 (tolérance aux fautes) est 800 fois supérieure à la probabilité d'existence dans le mode 3 (fautes non tolérées), ce qui montre l'avantage de la technique de protection. De l'autre coté, la probabilité d'existence dans le mode 2 (tolérance aux fautes) de l'instruction DUP est 63 fois supérieure à celle de l'instruction STORE. Ce résultat nous paraît logique dans la mesure où l'instruction STORE est plus complexe que l'instruction DUP et demande plus de ressources matérielles et de nombres de cycles.

# D. Généralisation de l'approche du flux informationnel sur tout le programme

## D.1. Programme de tri sans présence de la technique de protection

Notre principal objectif est d'évaluer la probabilité de défaillance de tout le programme. Pour ce faire, nous exploitons les probabilités de diverses instructions ainsi que celles de fonctions composant le programme de tri. La figure suivante présente l'arbre de défaillance du programme de tri sans prise en compte de la technique de protection.

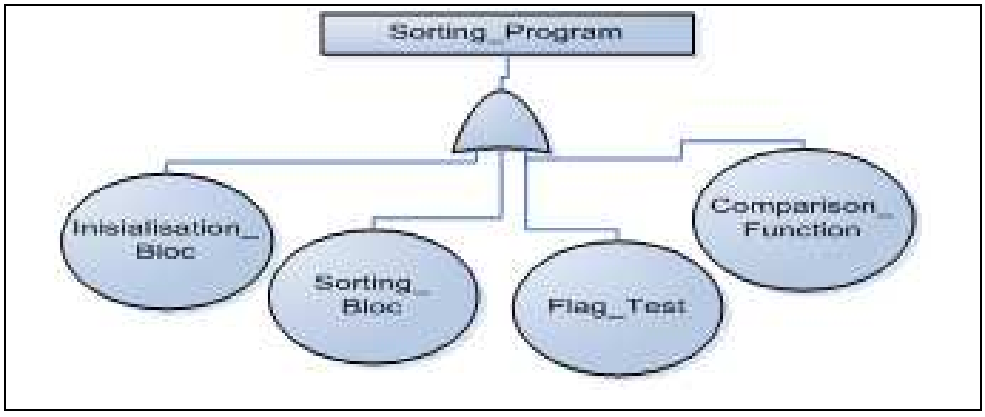


**Figure 4.10 : Arbre de défaillance du programme de tri sans présence de la technique de protection**

Le programme de tri est constitué de la fonction d'initialisation, de la fonction de comparaison, de la fonction de test du flag de permutation et du programme de la boucle principale effectuant le tri. La probabilité de l'élément au sommet est donc calculée à partir des probabilités des éléments de base. Ces éléments de base sont eux aussi des blocs et peuvent être dissociés eux même sous forme de nouveaux arbres de défaillance. À titre d'exemple, nous exposons dans la Figure 4.11 l'arbre de défaillance de la fonction de test du flag de permutation (*Flag_test*).

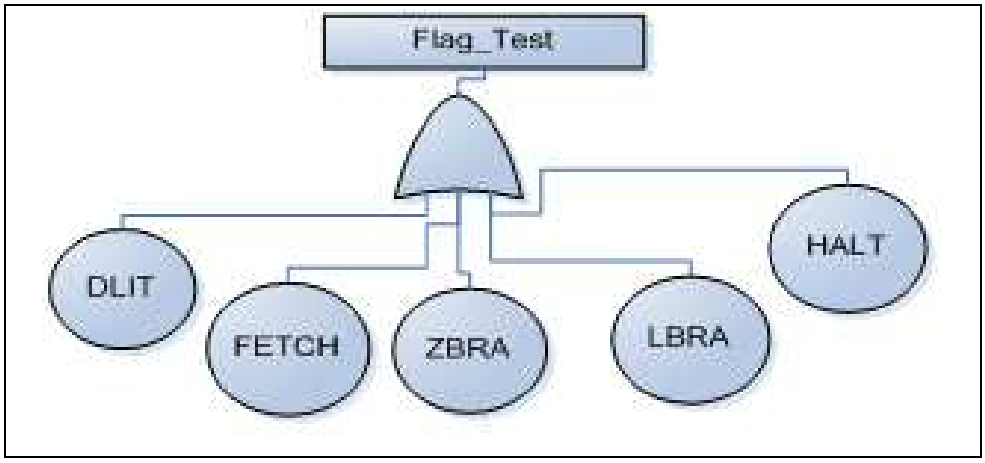


**Figure 4.11 : Arbre de défaillance de la fonction de test du flag de permutation ('Flag_test')**

Les instructions citées sur la figure sont les instructions que nous avons utilisées pour le développement de cette fonction de test du flag de permutation. Ainsi, à partir des probabilités des instructions DLIT, FETCH, ZBRA, LBRA et HALT, nous déterminons la probabilité de l'élément au sommet, à savoir du bloc '*flag_test*'. La démarche est pareille pour le reste des blocs de fonctions. Nous présentons les probabilités d'existence dans les modes 2 et 3 pour l'ensemble de fonctions composant le programme de tri [BELH08].

| | **Probabilité d'existence dans le** | |
|---|---|---|
| | **Mode 2 :** erreur détectée et tolérée | **Mode 3 :** erreur détectée et non tolérée |
| **Initialisation** | 0.0 | $3{,}091.\ 10^{-3}$ |
| **Boucle principale de tri** | 0.0 | $10{,}5.\ 10^{-3}$ |
| **Test du flag** | 0.0 | $4{,}59.\ 10^{-3}$ |
| **Comparaison** | 0.0 | $7{,}9.\ 10^{-4}$ |
| **Programme complet** | 0.0 | $6{,}67.\ 10^{-2}$ |

**Tableau 4.2 : Probabilités d'existence dans les deux modes 2 et 3 de fonctionnement**

Suivant le tableau 4.2, d'un coté, nous constatons que la probabilité d'existence dans le mode 2 (tolérance aux fautes) est nulle. Ceci est du à la non présence de la technique de protection. Dans la partie suivante, nous traitons le programme tri en prenant compte de la technique de protection.

## D.2. Programme de tri avec présence de la technique de protection

Nous intégrons dans cette partie la technique de protection pour le calcul des probabilités. Pour cela, il faut que nous déterminons les instructions utilisées pour le développement des fonctions supplémentaires ajoutées servant pour la sauvegarde et le recouvrement des éléments sensibles au cœur du processeur. Nous pouvons citer à titre

d’exemple, l’arbre de défaillance de la fonction de sauvegarde du pointeur de pile de données DSP.

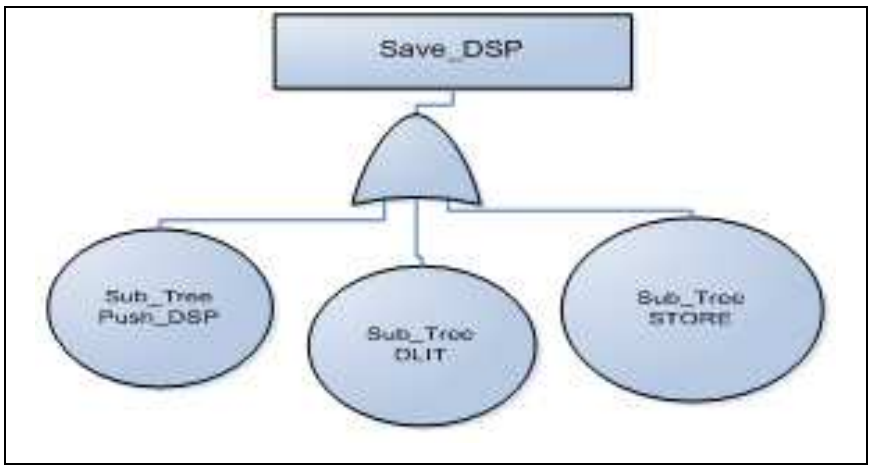


**Figure 4.12: Arbre de défaillance de la fonction de sauvegarde du pointeur de pile de données**

Cette fonction est composée des instructions Push_DSP, DLIT et STORE. Le traitement est encore de même pour les fonctions de sauvegarde et de recouvrement des autres éléments, à savoir le compteur programme, le sommet et le sous-sommet de la pile de données, le somment de la pile de retour et le pointeur de pile de retour. Ainsi, l’arbre de défaillance du programme de tri complet avec la prise en compte de la technique de protection est le suivant :

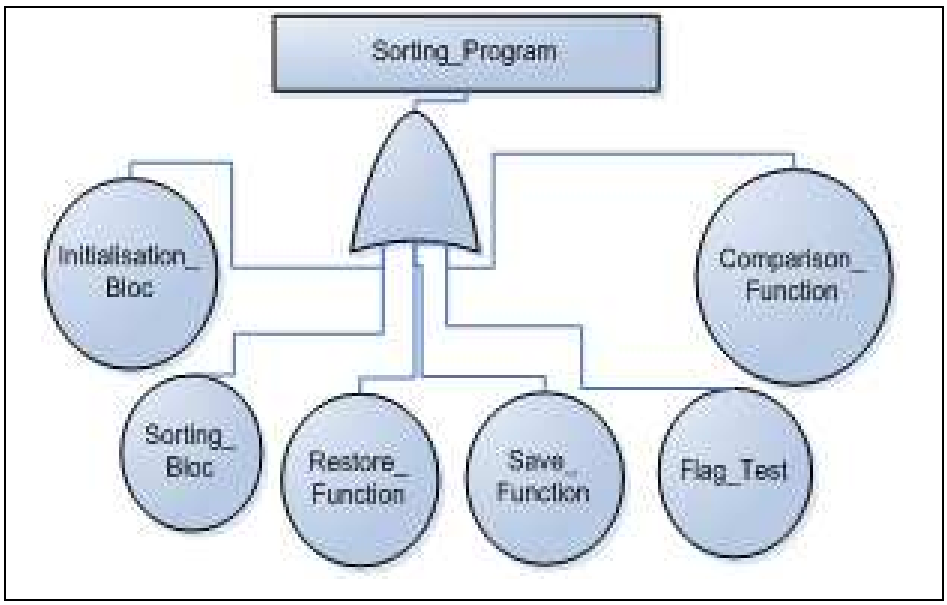


**Figure 4.13 : Arbre de défaillance du programme de tri avec présence de la technique de protection**

Nous présentons les probabilités d’existence dans les modes 2 et 3 pour l’ensemble de fonctions composant le programme de tri [BELH08].

| | Probabilité d'existence dans le | |
|---|---|---|
| | **Mode 2 :** erreur détectée et tolérée | **Mode 3 :** erreur détectée et non tolérée |
| **Initialisation** | $1{,}43.\ 10^{-2}$ | $3{,}091.\ 10^{-3}$ |
| **Boucle principale de tri** | $1{,}57.\ 10^{-2}$ | $10{,}5.\ 10^{-3}$ |
| **Test du flag** | $3{,}55.\ 10^{-3}$ | $4{,}59.\ 10^{-3}$ |
| **Comparaison** | $3{,}70.\ 10^{-2}$ | $7{,}9.\ 10^{-4}$ |
| **Fonction de sauvegarde** | $1{,}55.\ 10^{-2}$ | $2{,}2.\ 10^{-4}$ |
| **Fonction de restauration** | $0{,}33.\ 10^{-2}$ | $1{,}9.\ 10^{-5}$ |
| **Programme complet** | $1{,}57.\ 10^{-2}$ | $2{,}39.\ 10^{-4}$ |

**Tableau 4.3: Probabilités d'existence dans les deux modes 2 et 3 de fonctionnement**

Suivant le Tableau 4.3, nous constatons que la probabilité d'existence dans le mode 2 (tolérance aux fautes) n'est plus nulle et que la probabilité d'existence dans le mode 3 (non tolérance aux fautes) du programme complet a diminué d'un rapport de 1/279. Ceci est du à l'ajout des routines de protection, à savoir les fonctions de sauvegarde et de restauration des éléments sensibles de l'architecture. C'est-à-dire que si au départ nous avons 279 chances d'avoir une défaillance, en ajoutant les routines de protection nous aurions plutôt une seule chance. Comparons enfin les probabilités finales des deux programmes, que ça soit ils intègrent la fonction de tolérance ou pas, ce qui est résumé dans le Tableau 4.4 [BELH08].

| | Probabilité d'existence dans le | |
|---|---|---|
| | **Mode 2 :** erreur détectée et tolérée | **Mode 3 :** erreur détectée et non tolérée |
| **Programme complet sans tolérance** | 0.0 | $6{,}67.\ 10^{-2}$ |
| **Programme complet avec tolérance** | $1{,}57.\ 10^{-2}$ | $2{,}39.\ 10^{-4}$ |

**Tableau 4.4 : Comparaison des probabilités selon la présence ou l'absence des fonctions de recouvrement**

Nous constatons que lors de la prise en compte des fonctions de recouvrement, d'un coté la probabilité d'être dans le mode 2 (erreur détectée et tolérée) a augmenté et que de l'autre coté, celle d'être dans le mode 3 (erreur détectée et non tolérée) a diminué, ce qui confirme l'avantage de la technique de protection implantée.

Notre stratégie de protection a été évaluée et validée en premier lieu par l'émulateur en déterminant ses capacités de correction d'erreurs ainsi que ses surcoûts temporels selon divers scenarii d'erreurs. En second lieu, elle a été également évaluée et validée par l'approche du flux informationnel, ce qui nous a permis de conclure sur l'avantage de telle méthode de correction.

Arrivé à ce stade (proposition de méthodes d'évaluation et d'amélioration de la sûreté de fonctionnement d'une architecture de processeurs sûre de fonctionnement, et élaboration d'une démarche en vue d'une méthodologie globale de conception), nous nous proposons de définir à la fin de cette thèse quelques caractéristiques de l'architecture du processeur pipeliné à travers quelques détails supplémentaires permettant, pour un travail futur, le développement du modèle VHDL-RTL.

## E. Différents étages de l'architecture pipelinée

À partir du chemin de données de l'ensemble des instructions, nous établissons l'étage exécution du processeur pipeliné. En effet, le processeur contient deux étages différents : l'étage exécution et l'étage décodage. Un buffer d'instruction muni de son contrôleur sert à préparer l'instruction à être décodé. Nous schématisons dans la Figure 4.14 suivante une idée descriptive de ces deux étages, munis du buffer d'instructions avant de détailler chaque partie indépendamment dans la suite.

Au moment ou l'instruction k est entrain d'être exécutée, l'instruction suivante (k+1) est entrain d'être décodée et l'instruction (k+2) est dans le buffer d'instruction.

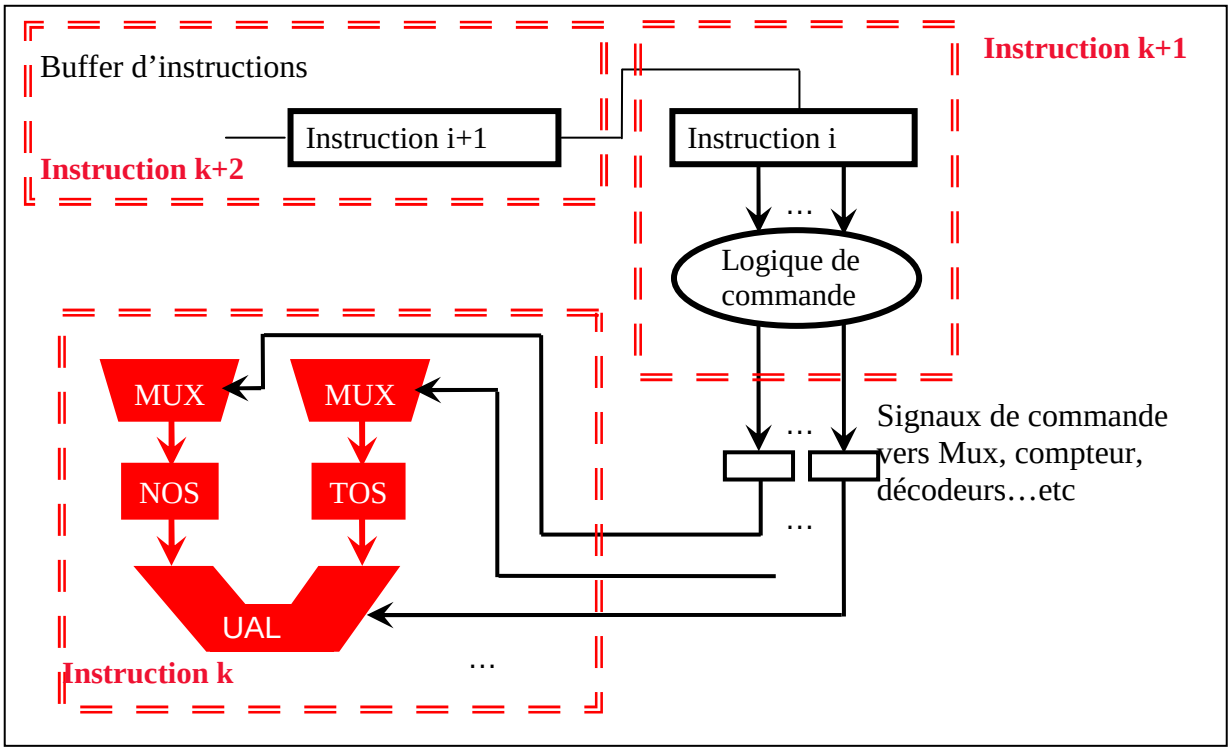


**Figure 4.14 : Pipeline deux étages du processeur munis du buffer d'instructions**

- ***Buffer et contrôleur d'instruction :***

Nous remarquons que sur 38 instructions, six instructions seulement ont besoin d'un opérande à la suite de leur code opération. En effet, les instructions LIT, SBRA, ZBRA ont besoin d'un opérande d'un octet :

— LIT a (octet)

— SBRA d (d est un déplacement (octet))

— ZBRA d (faire le saut de d octets si TOS est nul)

Quant aux instructions qui requièrent des opérandes de 2 octets, nous citons DLIT, CALL et LBRA :

— DLIT a (empiler un mot de 2 octets, a est un mot)

— CALL a (faire un appel à une routine à l'adresse a, a est un mot de 2 octets)

— LBRA a (faire un long branchement vers l'adresse a, a est un mot de 2 octets)

De ce fait, nous remarquons qu'à chaque fois, nous pouvons lire du buffer d'instruction un, deux ou trois octets, ceci selon le code opération du premier octet à travers lequel nous pouvons déterminer la longueur de l'instruction avec son opérande. Par exemple, si le premier octet est un DLIT, l'opérande se trouve alors dans les deux octets suivants. Par contre, si le premier octet est un DUP, nous n'avons pas d'opérande et donc c'est l'instruction suivante qui se trouve dans le prochain octet. C'est pour cette raison qu'un contrôleur d'instruction s'avère nécessaire pour gérer le remplissage et le vidage du buffer. Le modèle de cet étage est schématisé par la figure suivante.

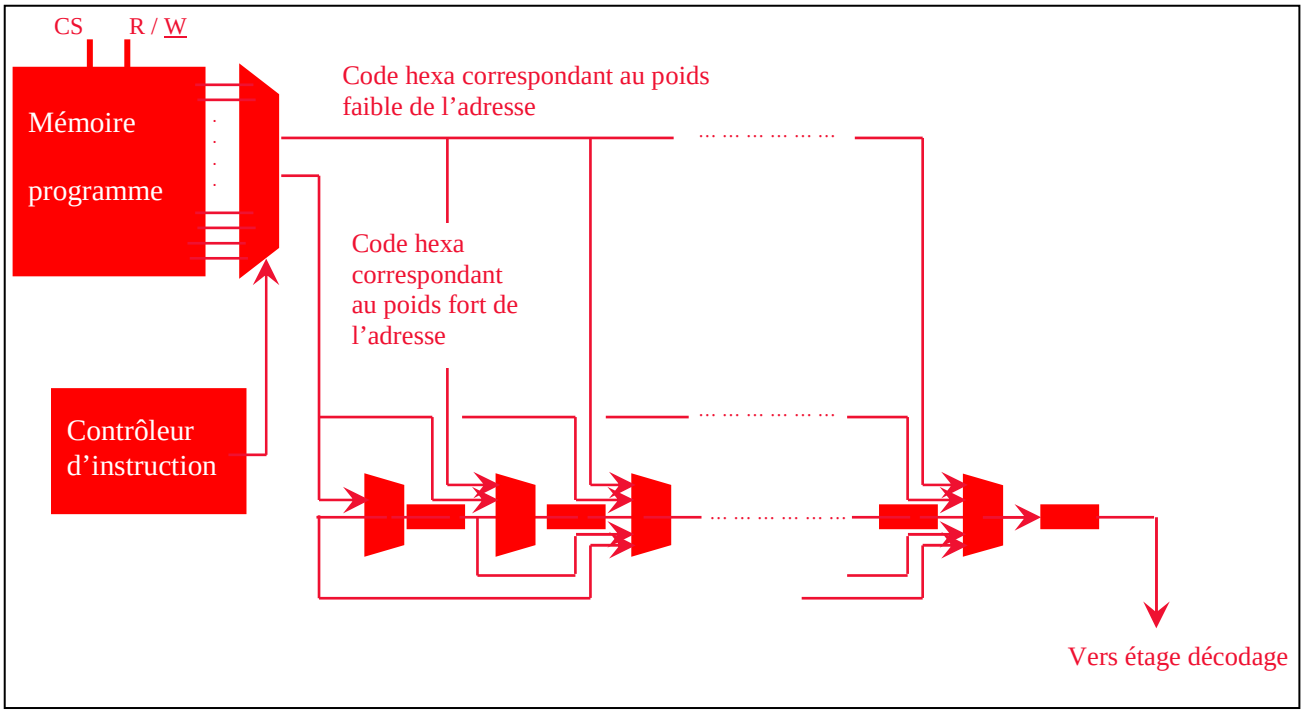


**Figure 4.15 : Caractéristiques architecturales de l'étage de bufferisation**

À partir de la mémoire programme, la lecture se fait par 16bits, poids fort et poids faible séparés. C'est le contrôleur d'instruction qui gère les signaux de commandes des différents multiplexeurs précédents les registres du buffer. Ça dépend entre autres des registres libres, de la taille de l'instruction en cours et de la nature de l'instruction elle-même (cas des instructions de saut). Pour chaque registre du buffer, il peut récupérer :

- Le code hexadécimal correspondant au poids faible de l'adresse dans le cas ou ce registre est le premier registre vide du buffer.
- Le code hexadécimal correspondant au poids fort de l'adresse dans le cas ou ce registre est le second registre vide du buffer.
- Le code hexadécimal se trouvant dans le registre qui le précède dans le cas ou l'instruction qui vient de s'écrire dans le code opératoire est une instruction d'un octet.
- Le code hexadécimal se trouvant dans le registre qui le précède de deux fois dans le cas ou l'instruction qui vient de s'écrire dans le code opératoire est une instruction de deux octets.
- Le code hexadécimal se trouvant dans le registre qui le précède de trois fois dans le cas ou l'instruction qui vient de s'écrire dans le code opératoire est une instruction de trois octets.

- ***Étage de décodage d'instruction et étage exécution :***

Tous les signaux de commande des multiplexeurs, des mémoires (lecture/écriture, activation), le code opératoire de l'UAL sont gérés dans cet étage.

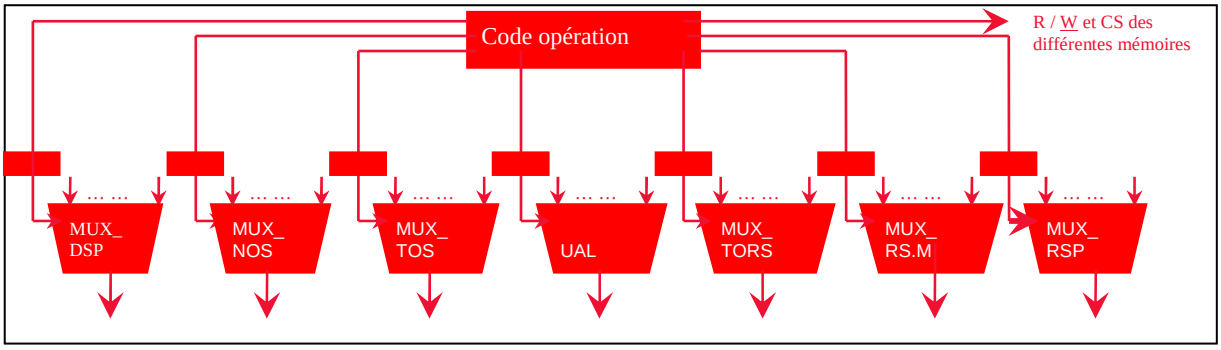


**Figure 4.16 : Caractéristiques architecturales de l'étage de décodage de l'instruction**

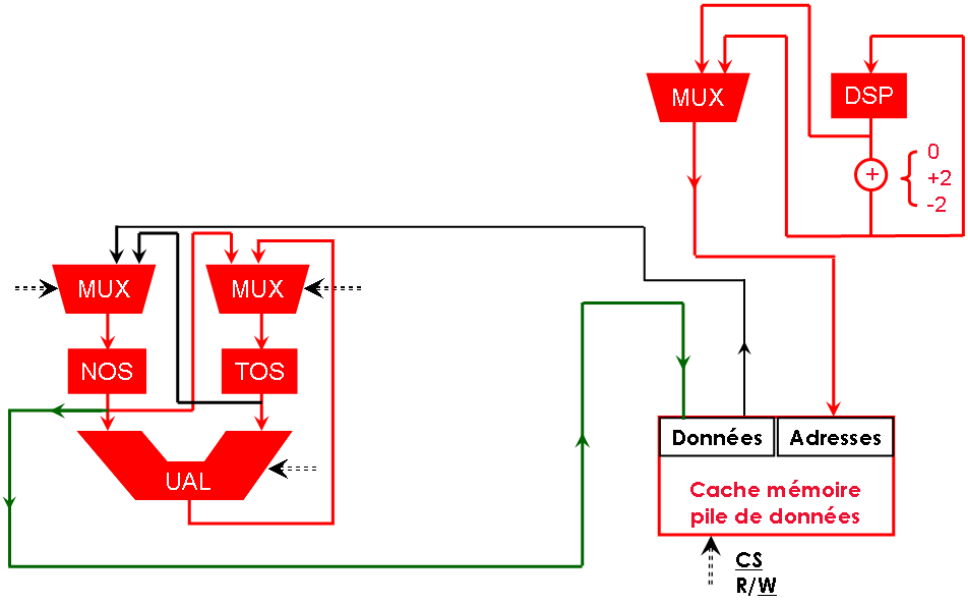


**Figure 4.17 : Caractéristiques architecturales de l'étage d'exécution**

En fait, cet étage est lié avec l'étage exécution dans la mesure où le premier prépare les signaux de contrôle de l'instruction en cours de décodage, pour qu'elle soit exécutée au cours du cycle suivant. L'exemple de la Figure 4.17 illustre un exemple de chemin de données. Les signaux de sélection des trois multiplexeurs ainsi que le signal de sélection de l'UAL sont précédemment prédéfinis lors du cycle précédent le cycle d'exécution, c'est-à-dire lors du cycle de décodage.

## F. Conclusions

Nous venons d'appliquer une approche permettant une évaluation probabiliste de l'ensemble architecture de processeur/application logicielle. Cette approche, fournie par l'équipe du CRAN de Nancy/A3SI-ENSAM de Metz, a servi à la validation d'une technique de correction logicielle implantée dans un *benchmark* tournant sous un émulateur de processeur. Cet émulateur a servi entre autres, pour simuler divers scenarii d'injection d'erreurs. Les schémas d'exécution et les chemins de données de certaines instructions ont été exposés dans ce chapitre afin de les exploiter pour l'élaboration des modèles de haut niveau de bas niveau de l'approche du flux informationnel. Ce qui nous a permis de déterminer les probabilités de présence dans un des modes de fonctionnement, incluant la détection des erreurs (mode 2 et mode 3). Par le calcul de ces probabilités, nous avons montré l'avantage de notre méthode de protection implantée dans le *benchmark*. Ainsi, nous avons établi une évaluation fiabiliste d'un ensemble application/processeur à l'aide de l'approche du flux informationnel. Cette méthode est également applicable indépendamment de l'application et indépendamment du processeur, ce qui répond aux exigences méthodologiques de cette thèse fixées par le projet CIM'Tronic.

Une fois que la méthode de protection a été validée, quelques caractéristiques de l'architecture pipelinée ont été présentées en vue du développement futur de l'architecture en VHDL-RTL.

# Références


— [BELH08] H. Belhadaoui : “ Conception sûre des systèmes mécatroniques intelligents pour des applications critiques”, Thèse de Doctorat en cotutelle entre l’Institut National Polytechnique de Lorraine (INPL) et l’École Nationale Supérieure d’Électricité et de Mécanique (ENSEM – Casablanca – Maroc) - spécialité Automatique et Informatique Industrielle, soutenue en 2008 au Maroc.

— [GROU95] Group Aralia: “Computation of prime implicants of a fault tree within Aralia”. In Proc. of The European Safety and Reliability Association Conference, ESREL’95, Bournemouth, UK, 1995: 190-202.

— [HAMI05] K. Hamidi, O. Malassé and J-F. Aubry : “Coupling of information-flow aggregation method and dynamical model for a more accurate evaluation of reliability”. In Proc. of The European Safety and Reliability Conference (ESREL05), Poland, June 2005.

— [IEC99] IEC 61508, 1999 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, Part 1-7. International Electrotechnical Committee.

— [JALL08a] M. Jallouli, H. Belhadaoui, C. Diou, F. Monteiro, A. Dandache, O. Malassé, G. Buchheit, J-F. Aubry & H. Medromi : “Dependability Consequences of Fault-Tolerant Technique Integrated in Stack Processor Emulator using Information Flow Approach”, In Proc. Of The IEEE

International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS08), Tozeur, Tunisia, March 2008.

— [YUCH07] M. Yuchang, L. Hongwei and Y. Xiaozong : “Efficient Fault Tree Analysis of Complex Fault Tolerant Multiple-Phased Systems”. Tsinghua Science and Technology. ISSN 1007-0214 22/49 vol. 12, No. S1, July 2007 pp.122-127.

# Conclusion générale

Dans le cadre du projet CIM'tronic (projet fondation CETIM) dont le but global est la conception intégrée de systèmes mécatroniques sûrs de fonctionnement, l'objectif de ce travail était de proposer et concevoir des outils d'aide à l'évaluation de la sûreté en vue d'une méthodologie de conception d'une architecture de processeur sûre de fonctionnement, ceci à travers le développement d'outils et de mécanismes de tolérances aux fautes d'une part et la conception et la définition des caractéristiques architecturales du processeur d'autre part. En outre, dans un contexte de partenariat et de collaboration entre les laboratoires du consortium sollicité par le projet CIM'tronic, nous avons été amenés à faire converger nos travaux avec ceux de l'équipe du CRAN de Nancy/A3SI-ENSAM de Metz afin d'évaluer certains paramètres de sûreté de fonctionnement.

Notre premier travail consistait à définir la démarche adoptée. Il s'agit de développer un émulateur de processeur intégrant les résultats et contraintes obtenus suite à l'étude de différents critères liés à la conception de l'architecture, plus particulièrement le type d'application qui est dédié au contrôle et à la commande, l'aspect sûreté de fonctionnement de l'architecture et le besoin en jeu d'instructions. L'émulateur permet d'étudier le comportement fonctionnel du processeur sans entrer dans les détails de sa structure interne en termes d'étages de pipeline et de cycles d'horloges par exemple. Toutefois, le développement de l'émulateur doit être précédé par le choix du type d'architecture et de sa structure interne. De ce fait, nous nous sommes focalisés sur la justification du type d'architecture suite à une étude comparative de diverses architectures et des divers processeurs tolérants aux fautes. Parmi les architectures disponibles, nous citons les architectures classiques CISC ou RISC

ainsi que l'architecture MISC (*Minimal Instruction Set Computer*). Sachant que le paradigme de sûreté guidant notre recherche est la simplicité, le choix architectural s'est orienté vers l'architecture MISC du fait de ses avantages, notamment en termes de surface, de nombre minimal d'instructions et de longueur fixe des instructions. Ainsi, en tenant particulièrement compte de la sûreté de fonctionnement, nous avons choisi une architecture à pile pour sa simplicité et sa flexibilité. En effet, la simplicité architecturale permet de réduire la durée de conception et surtout de rendre plus facile la protection du processeur. D'une part, la restauration d'un état correct après l'occurrence d'une erreur est plus simple du fait de la faible quantité d'informations mémorisées dans un processeur MISC. D'autre part, elle permet d'envisager plusieurs techniques de protection, qu'elles soient basées sur la duplication ou la sécurisation des différents éléments constitutifs du cœur du processeur, des bus et des mémoires.

Outre l'étude du comportement fonctionnel du processeur grâce au développement de *benchmarks* représentatifs de programmes embarqués, l'émulateur nous a permis de simuler une injection de différents scenarii de fautes. Selon le scénario d'apparition d'erreurs, périodiques, aléatoires ou par salve, nous provoquons une interruption qui a pour rôle de faire appel à une fonction de recouvrement. En effet, nous avons intégré dans un *benchmark* de tri des routines supplémentaires permettant la sauvegarde périodique dans une mémoire sûre externe ainsi que le recouvrement des éléments sensibles internes au cœur du processeur à savoir, le compteur programme, le sous-sommet et le sommet de la pile de données, le sommet de la pile de retour et les pointeurs des deux piles. À partir de ces diverses scenarii d'apparition d'erreurs, nous avons étudié les capacités de correction ainsi que les conséquences temporelles d'une telle méthode de protection logicielle [JALL07]. Ainsi, connaissant certaines spécifications du cahier des charge, telles que les contraintes temps-réel et le taux d'erreurs de l'environnement où se situe le processeur, nous pouvons justifier l'efficacité de notre technique de protection et nous pouvons définir la limite à partir de laquelle le processeur ne remplit plus son rôle. Un autre avantage de cette technique de protection consiste en son indépendance du programme assembleur décrit dans le *benchmark*, qui la rend par conséquent indépendante de l'application. Il s'agit en effet de blocs logiciels indépendants, dont il suffit juste de modifier les adresses de retour. Par ailleurs, cette technique de correction est également indépendante du moyen de détection. Indépendamment de ce dernier, il suffit juste que cette détection informe le processeur par le biais d'une interruption externe. C'est une technique de protection utilisable quelque soit l'application et

quelque soit la méthode de détection sous réserve d'envoyer une interruption en cas d'apparition d'erreur. Ce qui confirme l'aspect méthodologique de cette protection.

Outre cela, une autre méthode pour valider et évaluer cette technique de protection consistait à appliquer l'approche du flux informationnel fournie par l'équipe de recherche du CRAN de Nancy/A3SI-ENSAM de Metz, en nous basant sur les chemins de données des instructions ainsi que leurs étapes d'exécution. Ceci s'inscrit parfaitement dans le contexte des exigences du projet CIM'tronic, à savoir : faire converger les travaux de divers partenaires du consortium en un travail commun. Nous avons préalablement modélisé les différents schémas d'exécution de certaines instructions types, telles que les instructions arithmétiques et logiques, une instruction exigeant un seul cycle d'horloge et exploitant la mémoire pile de données (instruction DUP) et une instruction exigeant deux cycles d'horloge et exploitant la mémoire explicite (qui est partagée avec la mémoire de la pile de retour). Nous avons parfaitement expliqué les diverses opérations permettant l'exécution de ces instructions, ce qui a pour conséquences de faciliter la modélisation des différentes instructions par l'approche du flux informationnel, et donc de définir successivement les modèles de haut niveau et ceux de bas niveau. En s'inspirant de la norme IEC 61508 [IEC99], [BELH08] a défini six modes de fonctionnement, ce qui a permis de regrouper les scénarii de défaillance du modèle de bas niveau de chaque instruction par mode et de transformer ces scenarii en arbres de défaillance [BELH08]. La probabilité de chaque élément au sommet est alors calculée, ce qui correspond à la probabilité de défaillances de l'instruction en question. Nous avons conclu que, pour un cas initial d'étude de deux instructions [JALL08a], généralisé dans un premier temps pour un ensemble de fonctions, et puis dans un second temps pour le programme complet [JALL08b], la probabilité d'être dans le mode 2 (défaillance détectée mais tolérée) devient plus importante et que la probabilité d'être dans le mode 3 (défaillance détectée mais non tolérée) diminue. Ceci est dû à l'ajout des fonctions de sauvegarde et restauration des éléments sensibles de l'architecture et à la prise en compte d'une technique de tolérance aux fautes. Cela signifie que le nombre défaillances détectées et tolérées (corrigées) devient plus important lors d'utilisation de la technique de tolérance aux fautes, ce qui est un résultat logique démontrant bien l'avantage de cette méthode de protection. Ainsi, nous avons établi une évaluation fiabiliste d'un ensemble application/processeur à l'aide de l'approche du flux informationnel. Cette méthode est également applicable indépendamment de l'application et indépendamment du processeur ce qui répond aux exigences méthodologiques de cette thèse fixées par le projet CIM'tronic.

Cette approche peut-être avantageusement étendue en rendant systématique le passage du chemin de données et du schéma d'exécution d'une instruction vers les modèles de haut niveau et de bas niveau de l'approche du flux informationnel. Ceci peut se faire à travers le développement d'un outil logiciel de génération automatique de modèles.

Les objectifs méthodologiques étant atteints – proposition de méthodes d'évaluation et d'amélioration de la sûreté de fonctionnement d'une architecture de processeurs sûre de fonctionnement, élaboration de démarche en vue d'une méthodologie globale de conception – nous avons donné à la fin de cette thèse quelques caractéristiques de l'architecture du processeur pipeliné permettant lors d'un travail futur le développement du modèle VHDL-RTL. D'un autre coté, d'autres *benchmarks* peuvent être développés afin d'avoir des études statistiques et des conclusions plus générales sur les capacités de correction et les surcoûts temporels de cette technique de protection. Comme il est également possible de proposer d'autres scenarii d'apparition d'erreurs et de les ajouter à l'émulateur.

# Références


— [BELH08] H. Belhadaoui : “ Conception sûre des systèmes mécatroniques intelligents pour des applications critiques”, Thèse de Doctorat en cotutelle entre l’Institut National Polytechnique de Lorraine (INPL) et l’École Nationale Supérieure d’Électricité et de Mécanique (ENSEM – Casablanca – Maroc) - spécialité Automatique et Informatique Industrielle, soutenue en 2008 au Maroc.

— [IEC99] IEC 61508, 1999 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, Part 1-7. International Electrotechnical Committee.

— [JALL07] M. Jallouli, C. Diou, F. Monteiro & A. Dandache : “Stack processor architecture and development methods suitable for dependable applications”, in Proc. 3rd International Workshop on Reconfigurable Communication Centric System-On-Chips (ReCoSoC’07), Montpellier, France, June 2007.

— [JALL08a] M. Jallouli, H. Belhadaoui, C. Diou, F. Monteiro, O. Malassé, J-F. Aubry, A. Dandache, G. Buchheit & H. Medromi : “Dependability Consequences of Fault-Tolerant Technique Integrated in Stack Processor Emulator using Information Flow Approach”, In Proc. Of The IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS08), Tozeur, Tunisia, March 2008.

— [JALL08b] M. Jallouli, C. Diou, F. Monteiro, A. Dandache, H. Belhadaoui, O. Malassé, G. Buchheit, J-F. Aubry & H. Medromi : “Evaluation of important reliability parameters using VHDL-RTL modelling and information flow approach”, In Proc. of The European Safety and Reliability Conference (ESREL08), Valencia, Spain, September 2008.

# Liste des figures

# Liste des tableaux

# Liste des acronymes

— THESAME : THESAME mécatronique et management : réseau technologique pour les entreprises en mécatronique, productique et management de l'innovation
— ABS : Anti Blocking System
— ESP : Electronic Stability Program
— CETIM : CEntre Technique des Industries Mécaniques
— A3SI : Automatismes et Simulation pour la Sécurité des Systèmes Industriels
— ENSAM : Ecole Nationale Supérieure des Arts et Métiers
— CRAN : Centre de Recherche en Automatique de Nancy
— UMR : Unité Mixte de Recherches
— EA : Equipe d'Accueil
— InESS : Institut d'Électronique, du Solide et des Systèmes
— LPM : Laboratoire de Physique des Matériaux
— LICM : Laboratoire Interfaces Capteurs & Microélectronique
— SEE : Single Event Effect : effet d'une particule isolée
— SEL : Single Event Latchup : latchup par une particule isolée
— SET : Single Event Transient : transition par une particule isolée
— SEU : Single Event Upset : perturbation par une particule isolée
— MBU : Multi Bit Upset
— SEFI : Single Event Functional Interrupt
— SHE : Single Hard Error
— SEGR : Single Event Gate Rupture

— SEB : Single Event Burn-out
— EDAC : Error Detection And Correction code
— ECC : Error Correction Code
— DMR : Dual Modular Redundancy
— TMR : Triple Modular Redundancy
— CDE : Codes Détecteurs d'Erreurs
— TSC : Totally Self Checking
— STC : Self Testing Checkers
— UART : Universal Asynchronous Receiver Transmitter
— CRC : Cyclic Redundancy Code
— LUT : Look-Up Table
— ESA : European Space Agency
— ESTEC : European Space research and Technology Centre
— FT : Fault Tolerance
— NOC : Network On Chip
— FPGA : Field-Programmable Gate Array : réseau de portes programmables
— CISC : Complex Instruction Set Computer
— RISC : Reduced Instruction Set Computer
— MISC : Minimal Instruction Set Computer
— SdF : Sûreté de Fonctionnement
— LIFO : Last In First Out
— UAL : Unité Arithmétique et Logique
— CP : Compteur Programme
— DS : Data Stack : pile de données
— RS : Return Stack : pile de retour
— DSP : Data Stack Pointer : pointeur de pile de données
— RSP : Return Stack Pointer : pointeur de pile de retour
— TOS : Top-Of the Stack : sommet de la pile de données
— NOS : Next-Of the Stack : sous-sommet de la pile de données
— TORS : Top-Of the Return Stack : sommet de la pile de retour
— T.e.m : Taux d'erreurs maximal
— S.t : Surcoût temporel

— S.t.max : Surcoût temporel maximal

— Mux_DSP : Multiplexeur en amont du pointeur de pile de données DSP

— Mux_NOS : Multiplexeur en amont du sous-sommet de la pile de données

— Mem_DS : Mémoire pile de données

# Résumé

De nos jours, les systèmes électroniques embarqués deviennent de plus en plus utilisés dans notre vie quotidienne et doivent être de plus en plus sûrs. En effet, certains systèmes tels que les systèmes mécatroniques fonctionnent dans des conditions environnementales les faisant disposés à des erreurs en raison des perturbations. Ainsi, les concepteurs doivent considérer des remèdes à de telles erreurs. Dans ce travail, un intérêt particulier est consacré à la sûreté de fonctionnement des architectures de processeur. La philosophie du processeur à pile a été choisie puisqu'elle est un bon compromis entre simplicité et efficacité. L'approche que nous avons proposée, évaluée et validée, est basée sur le développement et l'exploitation d'un émulateur logiciel du processeur. La sûreté de fonctionnement du processeur est assurée par une exploitation mixte de techniques de protection : une détection matérielle d'erreurs et une correction logicielle. La technique de correction est implantée dans des *benchmarks* et est validée dans l'émulateur à travers une simulation de différents scénarii d'apparition d'erreurs. Des divers paramètres sont évalués tels que les capacités de correction et les surcoûts temporels. Cette technique de correction est indépendante de l'application et des moyens de détection, ce qui confirme l'aspect méthodologique de cette démarche. Par ailleurs, dans le cadre de la collaboration sollicitée par le projet CIM'Tronic, nous avons convergé nos travaux avec ceux de l'équipe du CRAN de Nancy/A3SI de Metz en appliquant l'approche du flux informationnel sur le jeu d'instruction du processeur. Nous avons montré la capacité de cette approche d'évaluer la fiabilité de l'ensemble processeur/application. Cette approche est aussi applicable indépendamment de l'application et indépendamment du processeur ce qui répond parfaitement aux exigences méthodologiques de cette thèse fixées par le projet CIM'Tronic.

# Abstract

Nowadays, reconfigurable, effective and dependable systems are becoming increasingly attractive for many applications. Furthermore, these systems should be more and more dependable. Indeed, systems such as mechatronic or automatically controlled ones often work in harsh environmental conditions making them more prone to errors due to disturbances. Thus, designers should consider ways to protect them against such errors. In this work, a special interest is dedicated to processor architecture dependability as we consider processor-based systems. The stack computer philosophy has been chosen for the processor architecture in order to achieve a good trade-off between simplicity and effectiveness. Our approach to introduce and evaluate the dependability is based on the development and the use of a software emulator of the processor to be designed. Dependability of the processor is ensured through the collaborative use of hardware and software protection techniques: hardware error detection means on one hand and software error correction means on the other hand. The correction technique is validated on the emulator using different fault injection scenarios on representative benchmarks of the target applications. Different parameters are evaluated such as correction capability and impact on time performance. This correction technique is independent from the target application and from the detection means. Otherwise, as specified by the CETIM, we integrated our work with the CRAN Nancy/A3SI Metz one by applying the information flow approach on the processor instruction set. We showed the ability of this approach to evaluate the hardware-software architecture.