Test en ligne des systèmes à base de microprocesseur

Thierry Michel

THESE

présentée par

Thierry MICHEL

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 30 mars 1992)

Spécialité: **Microélectronique (Conception)**

TEST EN LIGNE DES SYSTEMES A BASE DE MICROPROCESSEUR

Date de soutenance: 5 mars 1993

Composition du jury :

Madame	Gabrièle SAUCIER	
Messieurs	Christian LANDRAULT	Président et rapporteur
	Jean ARLAT	Rapporteur
	Bernard COURTOIS	
	René DOUCET	
	Régis LEVEUGLE	

Thèse préparée au sein du Laboratoire Conception de Systèmes Intégrés

Avant propos

Je tiens à remercier tout d'abord Madame Gabrièle SAUCIER, Professeur à l'ENSIMAG et directrice du laboratoire Conception de Systèmes Intégrés pour m'avoir accueilli dans son laboratoire.

Je remercie ensuite tout particulièrement Monsieur Régis LEVEUGLE, Maître de Conférences, pour ses remarques et son soutien tout au long de la préparation de cette thèse.

Je remercie également :

Monsieur Christian LANDRAULT, directeur de recherches CNRS au LIRMM, d'avoir accepté d'être rapporteur de cette thèse et de me faire l'honneur de présider le jury,

Monsieur Jean ARLAT, chargé de recherches CNRS au LAAS, d'avoir accepté d'être rapporteur de cette thèse,

Monsieur René DOUCET, chef du service "Architecture et développement logiciel" à Télémécanique, pour avoir accepté de faire partie du jury et pour avoir mis en place et suivi l'étude pratique qui fait l'objet d'une partie de ce document,

Monsieur Bernard COURTOIS, directeur de recherches CNRS et directeur du laboratoire TIMA, pour avoir accepté de faire partie du jury.

Je tiens aussi à remercier tout particulièrement Messieurs Pascal CHAPIER et Nicolas MAURIN, de Télémécanique Valbonne, pour l'aide précieuse qu'ils m'ont apporté dans la réalisation du projet présenté dans cette thèse.

Je remercie enfin les membres du CSI pour l'agréable ambiance de travail et tous ceux qui, de près ou de loin, ont contribué aux travaux présentés dans cette thèse, en particulier Raphael ROCHET.

Résumé

Cette thèse traite de la vérification en ligne, par des moyens matériels, du flot de contrôle d'un système à base de microprocesseur. Une technique de compaction est utilisée pour faciliter cette vérification (analyse de signature). La plupart des méthodes proposées jusqu'ici imposent une modification du programme d'application, afin d'introduire dans celui-ci des propriétés invariantes (la signature en chaque point de l'organigramme est indépendante des chemins préalablement parcourus). Les méthodes proposées ici, au contraire, ont comme caractéristique principale de ne pas modifier le programme vérifié et utilisent un dispositif de type processeur, disposant d'une mémoire locale, pour assurer l'invariance de la signature. Deux méthodes sont ainsi décrites. La première est facilement adaptable à différents microprocesseurs et présente une efficacité qui la place parmi les meilleures méthodes proposées jusqu'ici. La seconde méthode a été dérivée de la première dans le but de diminuer la quantité d'informations nécessaire au test. Cette dernière méthode a été implantée sur un prototype d'unité centrale d'automate programmable (avec la société Télémécanique) et son efficacité a été évaluée par des expériences d'injection de fautes. Le coût d'implantation particulièrement faible dans le cas du prototype réalisé peut permettre d'envisager une évolution de celui-ci vers un produit industriel.

Mots clefs :

Test en ligne, Vérification du flot de contrôle, Analyse de signature, Méthodes DSM, Processeurs watchdog, Processeurs à test en ligne intégré.

Abstract

This thesis deals with on-line test methods using hardware devices dedicated to control flow checking in microprocessor based systems. A compaction scheme (signature analysis) is used to facilitate the verification. Most of the methods proposed up to now require a program modification to force invariant properties (the signature at each point in the flowchart is independent from the previously followed paths). On the contrary, the methods proposed here do not require any modification in the verified program and use a watchdog processor with a local memory to ensure the signature is invariant. Two such methods are presented. The first one can be easily adapted to different microprocessors and is one of the most efficient methods proposed up to now. The second method, derived from the previous one, has been proposed in order to reduce the amount of information required for test purpose. This latter method was implemented in a programmable logic controller CPU (with the Telemecanique company) and its efficiency has been evaluated through fault injection experiments. The low overhead induced by the implemented mechanisms leads to envisage the evolution of the prototype towards an industrial product.

Key Words:

On-line test, Control flow checking, Signature analysis, Disjoint signature monitoring, Watchdog processors, Processors with built-in on-line test.

Table des matières

Avant propos 2
Résumé 3
Abstract 4
Introduction générale 1
Chapitre1. La vérification en ligne d'un flot de contrôle par analyse de signature ... 5
1.1. Généralités 6
1.1.1. Définitions : Graphe du Flot de Contrôle [Leve 90a] 6
1.1.2. Principe de la vérification d'un flot de contrôle 8
1.2. Fonctions et dispositifs de compaction 9
1.2.1. Choix d'une fonction de compaction 9
1.2.2. Dispositifs pour compaction parallèle par division polynomiale 10
1.2.2.1. MISR à OU Exclusifs internes 10
1.2.2.2. Probabilités de masquage d'une compaction par MISR 13
1.2.2.2.1. Influence entre erreurs 13
1.2.2.2.2. Modèle d'indépendance des erreurs 14
1.2.2.3. Structures équivalentes à un LFSR 16
1.2.3. Conclusion sur le choix d'un dispositif de compaction 16
1.3. Niveau du test : programme ou graphe d'états 17
1.3.1. Test d'un microprocesseur au niveau programme 17
1.3.1.1. Signatures imposées 17
1.3.1.2. Signatures dérivées 18
1.3.2. Test d'un circuit au niveau graphe d'états 19
1.4. Test au niveau programme 19
1.4.1. Représentation d'un programme par un graphe 19
1.4.2. Invariance et test de la signature 22
1.4.3. Localisation et repérage des informations pour le test 23
1.4.4. Méthodes ESM 25
1.4.4.1. Méthodes ESM sans ajustements 25
1.4.4.1.1. Méthode PSA de base [Namj 82a] 26
1.4.4.1.2. Méthode SIS [Shen 83] 27
1.4.4.1.3. Utilisation d'une fonction de compaction monotone [Saxe 89] 29
1.4.4.1.4. Remarques sur les méthodes sans ajustements 30
1.4.4.2. Méthodes ESM avec ajustements 32
1.4.4.2.1. Méthodes ESM avec ajustements sur arcs 33
1.4.4.2.2. Méthodes ESM avec ajustements sur noeuds 36
1.4.4.2.3. Traitement des sous-programmes 36
1.4.4.2.4. Algorithmes de génération des références 38
1.4.4.2.5. Techniques de réduction de la latence de détection .. 43
1.4.4.3. Conclusion sur les méthodes ESM 45
1.4.5. Méthodes DSM 46
1.4.5.1. Méthode Cerberus-16 [Namj 83] 46
1.4.5.2. Méthode ASIS [Eife 84] 48
1.4.5.3. Méthode OSLC [Made 91a] 50
1.4.5.4. Vérification multi-processeurs 51
1.4.5.5. Conclusion sur les méthodes DSM 52
1.4.6. Conclusion sur le test au niveau programme 53
1.5. Test au niveau graphe d'états d'une machine câblée 55
1.5.1. Principe de la méthode 56
1.5.1.1. Introduction d'une propriété invariante dans le graphe [Leve 90a] 56
1.5.1.2. Principe de la vérification 57

1.5.2. Implantation de contrôleurs S-codés 58
1.6. HSURF32 : un exemple de vérification à deux niveaux 60
1.6.1. Introduction 60
1.6.2. Test au niveau programme avec moniteur intégré au processeur 60
1.6.2.1. Génération d'une signature dérivée 61
1.6.2.2. Invariance et test de la signature 64
1.6.3. Test au niveau graphe d'état du contrôleur 64
1.6.4. Implantation du circuit 66
1.7. Conclusion 68
Chapitre 2. WDP : une méthode DSM modulaire 71
2.1. Introduction 71
2.2. Description de la méthode 72
2.2.1. Principe et origine de la méthode 72
2.2.1.1. Repérage des singularités par leur adresse 72
2.2.1.2. Vérification directe du séquencement du processeur 72
2.2.1.3. Décorrélation des signatures intermédiaires 73
2.2.2. Constitution du programme de vérification 73
2.2.2.1. Informations relatives à un noeud 74
2.2.2.2. Utilisation d'une technique de hachage 74
2.2.2.3. Localisation des noeuds avec un champ réduit 76
2.2.2.4. Structure des instructions watchdog 77
2.2.3. Fonctionnement du watchdog 78
2.2.3.1. Fonctionnement sur une instruction de séquencement 79
2.2.3.2. Fonctionnement sur un point de jonction 79
2.3. Génération des références 79
2.3.1. Exemple de programmes vérifiés 80
2.3.2. Algorithme de création du programme de vérification 81
2.3.3. Optimisations du programme de vérification 83
2.4. Capacité de détection d'erreur 85
2.4.1. Erreurs de bits 85
2.4.2. Erreurs de séquencement 87
2.4.3. Erreur de bits et de séquencement cumulées 88
2.4.4. Exemple numérique 89
2.4.5. WDP comparé à d'autres méthodes 89
2.4.5.1. Comparaison avec les méthodes sans ajustements 89
2.4.5.2. Comparaison avec les méthodes ESM avec ajustements 90
2.4.6. Conclusion sur le pouvoir de détection 91
2.5. Calcul du coût mémoire 92
2.5.1. Coûts mémoire minimums 93
2.5.2. Coûts mémoire à latence égale 96
2.5.3. Conclusion sur le coût mémoire de WDP 98
2.6. Prise en compte des caractéristiques du processeur vérifié 98
2.6.1. Processeur avec MMU intégrée 99
2.6.2. Processeurs avec lecture anticipée des instructions 100
2.6.2.1. Définitions 101
2.6.2.2. Cohérence de la signature en cas de branchement 102
2.6.2.3. Présence de plusieurs singularités dans la file d'attente du processeur 103
2.6.2.4. Adaptation de WDP aux processeurs pipelines 104
2.6.2.4.1. Fonctionnement 104
2.6.2.4.2. Exemple de singularités consécutives 108
2.6.2.4.3. Pipelines asynchrones et détection d'erreurs 109
2.6.3. Traitement des Exceptions 110

2.6.3.1. Reconnaissance d'une exception 110
2.6.3.2. Traitements associés au départ 111
2.6.3.3. Traitements associés au retour d'exception 112
2.6.4. Conclusion 113
2.7. Implantation d'un processeur watchdog WDP 113
2.7.1. Architecture Modulaire 114
2.7.1.1. Module Interface Microprocesseur 114
2.7.1.2. Module Compacteur 115
2.7.1.3. Module Exécuteur 116
2.7.1.4. Module Interface Mémoire 116
2.7.2. Application au microprocesseur Intel 80386sx 117
2.7.2.1. Caractéristiques du microprocesseur Intel 80386sx 117
2.7.2.2. Implantation du processeur watchdog 117
2.7.3. Application au microprocesseur MC68000 118
2.7.3.1. Caractéristiques du MC68000 119
2.7.3.2. Prise en compte du pipeline 119
2.7.3.3. Prise en compte des exceptions 121
2.7.4. Modélisation et simulation d'un watchdog WDP pour MC68000 122
2.7.5. Conclusion sur l'implantation de la méthode WDP 123
2.8. Conclusion sur la méthode WDP 124
Chapitre 3. DJAM : une méthode DSM avec ajustements 125
3.1. Introduction 125
3.2. Description de la méthode 125
3.2.1. Origine de la méthode 125
3.2.2. Principe de fonctionnement 127
3.2.3. Invariance des sous-programmes 129
3.2.4. Test de la signature 133
3.2.5. Structure des informations dans la table des références 134
3.2.6. Fonctionnement du moniteur par type de singularité 135
3.2.6.1. Branchements 135
3.2.6.2. Appels et Retours de sous-programme 136
3.2.6.3. Test de la signature 138
3.3. Génération des informations de test 138
3.3.1. Principe et limitations 138
3.3.2. Algorithme de calcul 139
3.4. Capacité de détection d'erreur 142
3.4.1. Erreurs de bits 143
3.4.2. Erreurs de séquencement 143
3.4.3. Conclusion 144
3.5. Calcul du coût mémoire 145
3.5.1. Coûts mémoire minimums 145
3.5.2. Coûts mémoire à latence égale 146
3.6. Prise en compte des caractéristiques du processeur 147
3.6.1. Exceptions 148
3.6.2. Pipeline 148
3.6.2.1. Pipeline synchrone 148
3.6.2.2. Pipeline asynchrone 148
3.7. Conclusion sur la méthode 149
Chapitre 4. Implantation de dispositifs d'analyse de signature 151
4.1. Objectifs de l'étude 151
4.2. Architecture générale d'une UC 152
4.2.1. Architecture matérielle 152
4.2.2. Exécution de l'application 153

4.2.2.1. Cycle d'une tâche 154
4.2.2.2. Exécution du code application d'une tâche 155
4.2.3. Dispositifs de test en ligne de base 156
4.2.3.1. Moyens matériels 156
4.2.3.2. Moyens logiciels 157
4.2.3.3. Exceptions du microprocesseur 80386 158
4.3. Vérification du flot de contrôle d'une UC 159
4.3.1. Principe du test 159
4.3.2. Analyse de signature sur le code application 160
4.3.2.1. Génération de la signature 160
4.3.2.2. Implantation de la méthode DJAM 160
4.3.2.3. Test de la signature 161
4.3.2.4. Génération des informations de test 162
4.3.3. Test au niveau système 163
4.3.3.1. Application du S-codage à la vérification d'un système 164
4.3.3.2. Stockage et accès aux références 165
4.3.3.3. Génération des références 166
4.3.4. Sécurité sur détection d'erreur 166
4.3.5. Fonctionnement multi-tâches 168
4.3.5.1. Changement de contexte 168
4.3.5.2. Gestion des références 168
4.4. Implantation des dispositifs de test 169
4.4.1. Organisation générale des dispositifs implantés 169
4.4.2. Implantation matérielle 170
4.4.2.1. Dispositif de génération de signature 171
4.4.2.2. Dispositif de contrôle de l'analyseur de signature 173
4.4.2.3. Dispositif de blocage des E/S 174
4.4.3. Implantation logicielle 175
4.5. Efficacité des mécanismes implantés 176
4.5.1. Modélisation d'un taux de détection global au niveau d'une UC 177
4.5.1.1. Principe de la modélisation 177
4.5.1.2. Résultats de la modélisation 181
4.5.2. Description des expériences d'injection de fautes 183
4.5.2.1. Description de l'injecteur de fautes DEF Injector 184
4.5.2.2. Déroulement des mesures 186
4.5.3. Injections de fautes sur le code application 189
4.5.3.1. Résultats des mesures 192
4.5.3.2. Interprétation 193
4.5.4. Injections de fautes sur le code système 195
4.5.4.1. Tableaux récapitulatifs des résultats de mesures 196
4.5.4.2. Amélioration apportée par l'analyse de signature 197
4.5.4.3. Efficacité des mécanismes de base 199
4.5.4.3.1. Chien de garde temporel et privilèges 386 199
4.5.4.3.2. Parité sur UCRF et UCAS (80386) 201
4.5.4.3.3. Parité sur UC86 202
4.5.4.4. Synthèse des résultats pour les erreurs injectées dans le système 202
4.5.4.4.1. Détection des erreurs par mécanisme 202
4.5.4.4.2. Répartition des erreurs potentiellement perturbatrices 204
4.5.4.5. Mesures complémentaires 205
4.5.4.5.1. Erreurs d'Entrées/Sorties 205
4.5.4.5.2. Erreurs détectées par l'application 206

4.5.5. Comparaison des mesures avec la modélisation du taux de détection 207
4.5.6. Conclusion sur l'efficacité 208
4.6. Conclusion sur l'implantation 209
Conclusion générale 211
Références bibliographiques 213
ANNEXES 221
1. Fonctionnement du watchdog WDP pour chaque type de noeud 221
1.1. Initialisation du processeur watchdog 221
1.2. Début de bloc linéaire 222
1.3. Branchement inconditionnel 222
1.4. Branchement conditionnel 223
1.5. Appel et retour inconditionnels de sous-programme 224
1.6. Appel et retour conditionnels de sous-programme 226
2. WDP et les structures HLL classiques 228
2.1. Structure "If Then" 228
2.2. Structure "If Else" 228
2.3. Structure "While_For" 228
2.4. Structure "Repeat Until" 228
2.5. Structure. "Switch" 229
2.6. Structure "Call" 229
2.7. Structure "Return" 229
3. Processeur WDP pour MC68000 230
3.1. Caractéristiques générales du MC68000 230
3.2. Le Processeur Watchdog 231
3.2.1. Schéma général et signaux 231
3.2.2. Synchronisation des modules 233
3.3. Description des Modules 235
3.3.1. Module Exécuteur 235
3.3.2. Module Interface Mémoire 236
3.3.3. Module Compacteur 238
3.3.4. Module Interface Microprocesseur 238
3.4. Exemple d'exécution d'une instruction WDP 240
3.5. Modèle VHDL du watchdog WDP pour MC68000 243

Introduction générale

La sûreté de fonctionnement est définie comme "la propriété permettant aux utilisateurs de placer une confiance justifiée dans le service délivré" [Lapr 85], [Lapr 92]. Le mauvais fonctionnement d'un système, ou défaillance, correspond à une différence entre le service rendu et le service spécifié. Une défaillance survient à la suite d'une erreur (partie de l'état du système déviant du comportement attendu). La cause de cette erreur est appelée une faute. Une erreur est donc la manifestation d'une faute dans le système, alors qu'une défaillance est l'effet d'une erreur sur le service.

Pour améliorer la sûreté de fonctionnement d'un système, il existe différents moyens :

- éviter qu'une faute soit présente dans le système,
- maîtriser les erreurs (détection et traitement) avant qu'elles ne produisent une défaillance du système.

La première solution utilise en particulier les techniques d'accroissement de la fiabilité des constituants du système. Dans le cas des systèmes informatiques, au niveau matériel, ceci peut être réalisé par la diminution du nombre de composants grâce à l'intégration.

Cependant, aussi efficaces soient elles, les techniques d'évitement des fautes ne sont pas toujours suffisantes. Il faut alors considérer que l'occurrence d'une faute dans un composant, ou l'activation d'une erreur de conception (dans le logiciel ou dans un composant VLSI, en particulier ASIC) sont des événements naturels dans le fonctionnement d'un système. Ces événements doivent alors donner lieu à un traitement spécifique si l'on veut en éviter les conséquences. Une technique de tolérance, par masquage des erreurs, ou par détection puis traitement des erreurs, permet ainsi d'éviter une défaillance du système en présence de certaines erreurs.

Les techniques habituelles de masquage d'erreurs ont recours à la redondance massive (TMR ou NMR[1]). Elles font appel à un vote majoritaire pour déterminer les sorties du système. La détection d'une erreur n'altère donc pas la capacité opérationnelle du système tant que la redondance est suffisante pour tolérer cette erreur. Après détection et localisation d'une erreur, une opération de maintenance peut avoir lieu pour restituer au système ses propriétés originales. Ces techniques,

---

[1] TMR : Triple Modular Redundancy, NMR : N Modular Redundancy.

qui introduisent un fort coût en matériel (redondance structurelle) ne seront pas considérées dans cette thèse.

Dans les techniques de détection puis de traitement d'erreur, la latence de détection joue un rôle essentiel. Lorsque la continuité du service est la caractéristique recherchée, la latence de détection détermine l'aptitude du système à revenir à un fonctionnement normal par reprise de processus. Lorsque la sécurité du système est visée, une latence réduite permet d'éviter une défaillance du système avant la détection. Par ailleurs, la latence de détection doit toujours être aussi faible que possible pour éviter une accumulation d'erreurs qui pourrait poser des problèmes de masquage pour certains mécanismes de détection.

Dans un schéma de tolérance aux fautes, la détection des erreurs est donc une étape essentielle de la sûreté de fonctionnement. Nous nous intéressons ici aux méthodes de test en ligne permettant la vérification continue du bon fonctionnement d'un système alors même qu'il est opérationnel, et à travers son mode de fonctionnement normal.

Les techniques de test en ligne considérées dans cette thèse excluent tout recours à la redondance massive (duplication) et permettent de détecter, avec un coût réduit, une certaine proportion de fautes, aussi bien permanentes que transitoires. Ces techniques sont soit destinées à des applications à faible exigence sécuritaire, soit utilisées en complément de la redondance massive, par exemple pour diminuer la latence de détection ou pour introduire des possibilités de diagnostic dans un système redondant. Pour un système à base de microprocesseur, les méthodes permettant une vérification du flot de contrôle sont particulièrement efficaces [Schm 82], [Schu 86], [Gunn 89], [Czec 90]. Afin de conserver un coût faible, une technique de compaction d'information est en général utilisée. L'objet de cette thèse est la proposition de deux nouvelles méthodes de vérification d'un flot de contrôle pour des systèmes à base de microprocesseur, ainsi que la présentation de leur mise en œuvre. Ces deux méthodes permettent d'effectuer la vérification sans modification des programmes d'application exécutés par le microprocesseur.

Dans le premier chapitre, après une définition du principe et des dispositifs de compaction utilisés, les principales méthodes de vérification d'un flot de contrôle sont abordées. Un exemple de microprocesseur est décrit à la fin de ce chapitre. Il dispose de mécanismes de test en ligne intégré permettant une vérification à travers l'exécution de son programme, et une vérification du flot de contrôle au niveau du séquenceur du circuit.

Dans le second chapitre, une première méthode originale de vérification du flot de contrôle d'un microprocesseur est étudiée (méthode WDP). Cette méthode

présente comme caractéristiques principales une très grande facilité d'adaptation à différents microprocesseurs et une transparence d'utilisation quasi totale. De plus, elle figure parmi les méthodes les plus efficaces, en terme de détection, présentées jusqu'à maintenant.

Dans le troisième chapitre, une seconde méthode pour la vérification du flot de contrôle d'un microprocesseur est présentée (méthode DJAM). Il s'agit d'un dérivé de la méthode précédente permettant de réduire le volume d'informations nécessaire à la vérification. Il s'agit aussi de la première méthode proposée permettant d'utiliser les techniques d'ajustements de signature sans modifier le programme exécuté par le microprocesseur.

Le dernier chapitre présente une étude effectuée en collaboration avec la société Télémécanique, et la réalisation d'un prototype d'unité centrale d'automate programmable intégrant des moyens de test en ligne pour la vérification du flot de contrôle de l'automate. Les principaux dispositifs implantés sur ce prototype permettent la mise en œuvre de la méthode DJAM. Des expériences d'injections de fautes ont permis de vérifier l'amélioration du niveau de sûreté par comparaison avec une unité centrale dépourvue de vérification de flot de contrôle.

Chapitre1. La vérification en ligne d'un flot de contrôle par analyse de signature

Le test en ligne continu d'un système simplex comprend de manière générale la surveillance de cet élément, pendant son fonctionnement normal, par un dispositif spécifique que nous nommerons "watchdog" s'il s'agit d'un processeur lui-même contrôlé par un programme ou "moniteur" dans le cas contraire. Suivant le niveau d'abstraction du système auquel à lieu la vérification, l'élément contrôlé peut être un système à base de microprocesseur ou un circuit VLSI complexe. Le dispositif de test est conçu pour vérifier certaines propriétés caractéristiques du bon fonctionnement de l'élément sous test. Du bon choix de ces propriétés dépend l'efficacité du test en ligne, c'est-à-dire la proportion d'erreurs détectées et la latence de détection, ainsi que le coût en matériel et en performances.

Une bonne introduction aux principales méthodes de test en ligne à faible coût pour les systèmes à base de microprocesseurs peut être trouvée dans [Mahm 88]. Les propriétés vérifiées se rapportent en général :

- à l'application traitée : test de vraisemblance ou de la plage de variation de certaines variables [Mahm 83], [Mahm 85a], codage des informations et des calculs [Mart 90],
- au logiciel d'application : vérification des branchements [Lu 82], [Namj 82a], ou vérification de propriétés temporelles par chien de garde [Conn 72],
- au matériel supportant l'application : abstraction du fonctionnement [Ayac 79], autorisations d'accès à certaines zones mémoire [Namj 82b], détection des codes opératoires illégaux d'un processeur.

Il a été montré dans [Schm 82], [Gunn 89] et [Czec 90] que, dans le cas d'un microprocesseur, les propriétés concernant le séquencement de l'élément sont les plus judicieuses à considérer. Ce chapitre est donc consacré à la description des principales méthodes de test en ligne continu d'un flot de contrôle. Les autres types de vérification, bien que très efficaces pour certaines, ne seront abordés ni dans ce chapitre ni dans cette thèse [Ayac 79], [Namj 82a], [Mahm 83], [Mart 90].

Deux niveaux de vérification d'un flot de contrôle seront définis dans la section 1.3 (niveau programme et niveau graphe d'états). Chacun d'eux sera plus particulièrement développé aux sections 1.4 et 1.5 et un exemple de vérification aux deux niveaux sera présenté en section 1.6 . Toutefois, il est nécessaire d'introduire au préalable un certain nombre de concepts et de définitions communes à l'ensemble des méthodes. Ceci sera fait en section 1.1 . La section 1.2, quant à elle,

permettra d'introduire les méthodes de compaction d'information utilisées aux différents niveaux de test.

1.1. Généralités

1.1.1. Définitions : Graphe du Flot de Contrôle [Leve 90a]

Un flot de contrôle peut être représenté, indépendamment de la nature de l'élément contrôlé, par un graphe orienté G = { $\mathbb{N}$, $\mathbb{A}$ }. Ce graphe sera dénommé par la suite GFC (Graphe du Flot de Contrôle). $\mathbb{N}$ est l'ensemble des noeuds du graphe et $\mathbb{A}$ est l'ensemble des arcs orientés représentant les transitions autorisées entre les noeuds de $\mathbb{N}$.

A chaque noeud sont associés :

- une information permettant d'identifier le noeud de façon unique dans le GFC,
- une information indiquant quelle opération doit être effectuée sur ce noeud (éventuellement aucune).

A chaque arc sont associés :

- un noeud origine No,
- un noeud destination Nd,
- un prédicat qui détermine sous quelles conditions la transition doit être effectuée,
- une information indiquant quelle opération doit être effectuée lors de la transition (éventuellement aucune).

L'ensemble des arcs définit la relation ∂ de succession entre états, telle que $\partial(No, P) = Nd$ si et seulement si il existe un arc de $\mathbb{A}$ de prédicat P ayant No comme noeud origine et Nd comme noeud destination.

Les méthodes de vérification du flot de contrôle font souvent intervenir des notions sur la structure des graphes, en particulier la notion de chemin (au sens classique de la théorie des graphes), et leur correction ou leur légalité.

Définition 1.1 : un chemin est correct dans un GFC si la relation ∂ de succession entre états est satisfaite pour l'ensemble des arcs du chemin. Dans le cas contraire, le chemin est incorrect.

Définition 1.2 : un chemin est légal dans un GFC si pour tout arc i, d'origine No_i et de destination Nd_i, du chemin considéré, il existe un prédicat P_i tel que $\partial(No_i, P_i) = Nd_i$. Dans le cas contraire, le chemin est illégal.

Un chemin légal est correct si toutes les transitions sont effectuées alors qu'elles sont effectivement autorisées, donc si les bons prédicats sont associés aux différents arcs du chemin.

Il existe une relation d'implication entre les notions de correction et de légalité d'un chemin :

- tout chemin correct est légal,
- un chemin légal est incorrect si un ou plusieurs prédicats associés aux arcs de ce chemin sont faux,
- tout chemin illégal est incorrect.

Pour décrire la structure d'un GFC, les notions de divergence, convergence, et point de jonction sont utilisées :

Définition 1.3 : une divergence est un ensemble d'arcs de $\mathbb{A}$ de même origine ayant des destinations différentes.

Définition 1.4 : une convergence est un ensemble d'arcs de $\mathbb{A}$ d'origines différentes ayant la même destination.

Définition 1.5 : un point de jonction est un noeud de $\mathbb{N}$ qui est la destination d'une convergence.

Les prédicats associés aux arcs d'une divergence sont supposés tous différents et non intersectants (pas de parallélisme dans le graphe[1]). Si le graphe est entièrement spécifié, ils forment de plus une tautologie.

Quelques autres termes se rapportant à la structure des graphes sont également utiles :

Définition 1.6 : un chemin linéaire est un sous graphe du GFC qui ne contient aucune divergence : si le premier noeud est atteint, alors tous les noeuds de ce chemin seront parcourus dans l'ordre indiqué par l'orientation des arcs.

Définition 1.7 : un chemin maximal est un chemin du GFC qui comporte au moins un noeud qui n'appartienne à aucun autre chemin du GFC ayant même noeud de départ et même noeud d'arrivée.

---

[1] Un formalisme tel que celui des réseaux de Pétri n'est donc pas nécessaire.

1.1.2. Principe de la vérification d'un flot de contrôle

La vérification d'un flot de contrôle consiste à comparer le flot de contrôle obtenu lors du fonctionnement du système, avec un GFC de référence. Une vérification complète consiste donc à s'assurer que :

- le chemin parcouru lors de l'exécution est correct dans le GFC de référence,
- les opérations effectuées sur chaque noeud (ou sur chaque transition) du chemin parcouru sont celles qui sont associées aux noeuds (ou aux transitions) de ce chemin dans le GFC de référence,
- les opérations commandées à l'élément sont exécutées correctement.

Pour réaliser ce test, il faut en pratique être capable d'extraire le flot de contrôle du système, d'avoir mémorisé le GFC de référence de ce système, et de disposer d'un mécanisme effectuant la comparaison. Ce test sera dit "en ligne" si cette vérification est de plus effectuée en temps réel, pendant le fonctionnement normal du système.

La quantité d'information relative à ce test est très importante. Pour éviter d'avoir à mémoriser un tel volume de données, il parait judicieux d'utiliser une technique permettant de représenter cette information sous une forme réduite. Cette réduction sera appelée une compaction et le résultat de cette compaction sera appelé une signature.

La vérification en ligne d'un flot de contrôle par analyse de signature consiste donc à générer une signature à partir du flot de contrôle obtenu en fonctionnement, puis à comparer ensuite cette signature avec une valeur de référence obtenue en appliquant la même fonction de compaction sur le GFC de référence. La fréquence des comparaisons détermine la latence minimale de détection d'une erreur par ce dispositif de test.

L'efficacité de cette vérification dépend essentiellement de la représentativité de la signature par rapport au flot de contrôle du système. Deux éléments peuvent ainsi altérer l'efficacité du test : d'une part la perte d'information liée à la fonction de compaction et, d'autre part la prise en compte d'une partie seulement des informations du flot de contrôle pour générer la signature.

La perte d'information entraînée par la fonction de compaction est appelée masquage et elle dépend du choix de la fonction de compaction (cf 1.2).

Le type des informations utilisées pour générer la signature dépend de la complexité de leur extraction lors du fonctionnement du système (problèmes d'observabilité), de la difficulté de génération des valeurs de référence à partir d'une description du système et du volume nécessaire au stockage des valeurs de

référence. Un compromis est donc à faire à ce niveau entre les possibilités ou le coût de réalisation du dispositif de vérification et le pouvoir de détection souhaité.

Il faut remarquer que la vérification d'un flot de contrôle par analyse de signature ne s'applique qu'à des systèmes pour lesquels il est possible de connaître le GFC de référence à partir d'une description du système. Ce GFC doit donc être invariable dans le temps et ne peut être modifié en cours d'exécution. En particulier, les programmes s'auto modifiant ne peuvent être vérifiés par les techniques qui vont être présentées ici et sont donc exclus a priori.

1.2. Fonctions et dispositifs de compaction

1.2.1. Choix d'une fonction de compaction

Le choix d'une fonction de compaction pour un dispositif d'analyse de signature est important car il détermine le taux de masquage entraîné par la compaction d'information. Par ailleurs, dans les techniques de test intégré, le dispositif de compaction se doit d'être particulièrement simple pour permettre un coût matériel aussi faible que possible. On s'intéressera ici uniquement à l'analyse de signature appliquée à la vérification d'un flot de contrôle et on ne parlera donc que de la compaction d'un flot d'information parallèle, c'est-à-dire de données se présentant sous forme de vecteurs de k bits en entrée du dispositif de compaction.

Les fonctions de compaction classiquement utilisées dans les schémas de vérification d'un flot de contrôle par analyse de signature sont :

- la somme modulo 2 (OU Exclusif bit à bit) [Namj 82a],
- la somme arithmétique [Saxe 89],
- la division polynomiale [Shen 83], [Wilk 87] ....

A quelques exceptions près qui seront mentionnées par la suite, le choix d'une fonction de compaction pour un schéma de vérification d'un flot de contrôle est libre et ne dépend que des considérations liées à la probabilité de masquage et au coût matériel.

D'un point de vue implantation matérielle, la somme modulo 2 est la plus économique.

La somme arithmétique requiert un nombre de bits de signature plus élevé que la taille des vecteurs à compacter, sauf si on utilise une somme modulo k. Dans les deux cas, cela nécessite un additionneur complet, avec une propagation de retenue qui peut entraîner des temps de compaction élevés.

Ces fonctions de compaction ont toutes une propriété particulière par rapport à une division polynomiale : la commutativité. Elles ne sont donc pas représentatives

de l'ordre dans lequel les données sont compactées, alors qu'avec une compaction par division polynomiale, l'ordre des données a une importance.

Si l'on considère de plus que les structures les plus simples des dispositifs de compaction par division polynomiale sont à peine plus coûteuses que celles pour une somme modulo 2, mais en présentant un masquage beaucoup plus faible, on comprend facilement pourquoi la division polynomiale est de loin la fonction de compaction la plus utilisée actuellement, non seulement pour la vérification d'un flot de contrôle, mais aussi et surtout dans les techniques de BIST (Built-In Self-Test) [Gels 87], [Harw 89]. C'est la raison pour laquelle la dénomination "analyse de signature" a souvent été prise comme synonyme de "analyse de signature par division polynomiale". Dans la suite de cette thèse, par contre, le terme "signature" sera utilisé indépendamment de la notion de division polynomiale.

De nombreuses études théoriques sur les possibilités de masquages ont été réalisées ces dernières années sur ce mode de compaction, essentiellement pour leur utilisation dans des dispositifs de BIST. Les principaux résultats seront présentés dans la section 1.2.3, après la description des dispositifs eux-mêmes.

1.2.2. Dispositifs pour compaction parallèle par division polynomiale

Pour une compaction par division polynomiale, le coût matériel dépend de la structure du dispositif et de la manière de réaliser le polynôme diviseur (câblé ou programmable). Un bon aperçu des différentes structures de ces dispositifs de compaction peut être trouvé dans [Leve 90a]. Pour la compaction d'un flot d'information parallèle, le dispositif est appelé un MISR (Multiple Input Shift Register). L'appellation LFSR (Linear Feedback Shift Register) sera réservée aux circuits de compaction par division polynomiale d'un flot d'information série.

Dans les dispositifs de compaction parallèle présentés dans [Leve 90a], on ne détaillera ici que la structure à OU Exclusifs internes (Figure 1.1). Cette structure présente des avantages sur les autres structures (OU Exclusifs externes, hybrides) d'un point de vue performance et d'un point de vue calcul des signatures attendues (signatures de référence). Les structures équivalentes à un LFSR seront toutefois mentionnées car elles offrent certains avantages mais ne sont que peu utilisées en raison de leur coût matériel plus important.

1.2.2.1. MISR à OU Exclusifs internes

Ce dispositif (Figure 1.1) est celui qui est le plus intéressant pour les raisons de performances et de facilité de calcul qui viennent d'être mentionnées.

Les performances temporelles de ce circuit sont optimales car le chemin critique[1] est très court (deux niveaux de portes) et il ne dépend ni du polynôme, ni de la manière de le réaliser (câblé ou programmé) ni du nombre de cellule du MISR. A l'inverse, un MISR à OU Exclusifs externes et polynôme programmable [Leve 90a] présente un chemin critique qui croît linéairement avec le nombre de cellules du MISR.

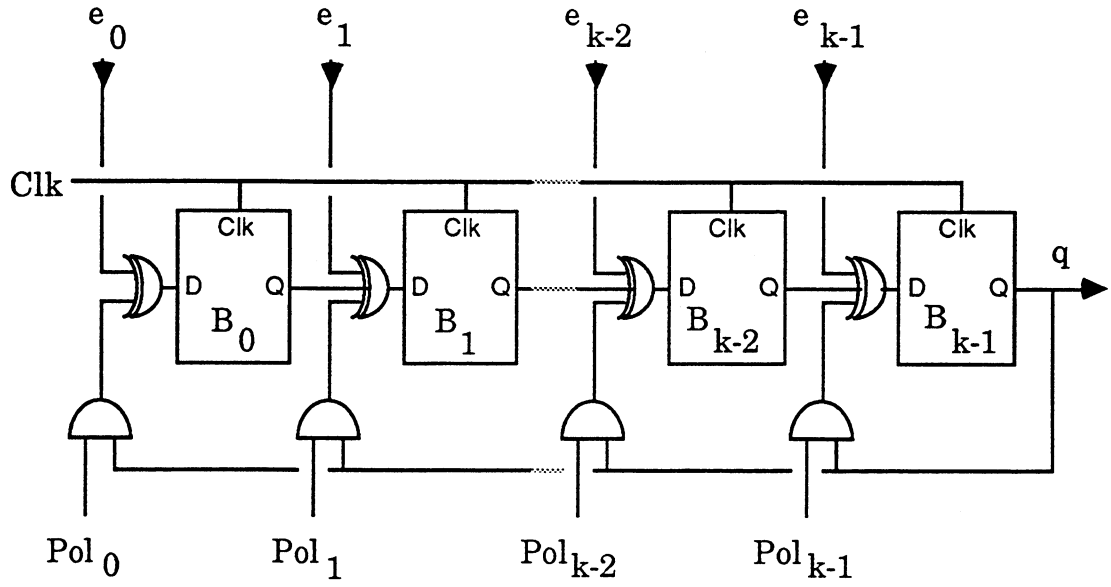


Figure 1.1 : MISR à OU Exclusifs internes avec polynôme programmable.

L'émulation d'un MISR par logiciel (pour le calcul des signatures de référence) est particulièrement simple avec la structure à OU Exclusifs internes, contrairement aux structures possédant des OU Exclusifs externes, car il n'est pas nécessaire de recréer la chaîne de rebouclage du dispositif en considérant individuellement chaque bit de la signature. L'algorithme est donc trivial ce qui permet de réduire le temps de calcul des signatures intermédiaires lors de la phase de génération des références. Un exemple de programmation est indiqué ci-dessous en langage C.

---

1 Chemin électrique sur lequel la propagation du signal d'entrée présente le temps le plus élevé (détermine la fréquence maximale du circuit).

```
unsigned short int   signature, /* signature sur 16 bits */
                     polynome,
                     donnee;

/* tester le bit 15 */
if (signature >= 0x8000) donnee = donnee ^ polynome;
signature = (signature << 1) ^ donnee;
```

Représentation matricielle

Le MISR représenté sur la figure 1.1 est régi par les équations suivantes :

$$b_i(t+1) = b_{i-1}(t) \oplus (Pol_i \,.\, b_{k-1}(t)) \oplus e_i(t) \qquad 1 \le i \le k-1$$
$$b_0(t+1) = (Pol_0 \,.\, b_{k-1}(t)) \oplus e_0(t)$$

$$q(t) = b_{k-1}(t)$$

où $b_i(t)$ représente la valeur de la sortie de la bascule B_i à l'instant t ($t \ge 0$).

La représentation matricielle du fonctionnement de ce circuit est :

$$\begin{pmatrix} b_0(t+1) \\ . \\ . \\ . \\ . \\ b_{k-1}(t+1) \end{pmatrix} = \begin{pmatrix} 0 & \cdots & 0 & Pol_0 \\ 1 & 0 & & Pol_1 \\ 0 & 1 & 0 & Pol_2 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & & \cdot 0 & \cdot \\ 0 & \cdots & 0\,1 & Pol_{k-1} \end{pmatrix} \begin{pmatrix} b_0(t) \\ . \\ . \\ . \\ b_{k-1}(t) \end{pmatrix} \oplus \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots 0 \\ \cdot & 0 & 1 & 0 \cdot 0 \\ \cdot & \cdot & \cdot & \cdot \cdot \cdot \\ \cdot & \cdot & \cdot & \cdot 1\,0 \\ 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} e_0(t) \\ . \\ . \\ . \\ e_{k-1}(t) \end{pmatrix}$$

ou encore :

$$B(t+1) = M\,B(t) \oplus C\,E(t)$$

B(t) : vecteur formé par les sorties des k bascules à l'instant t (état du MISR à l'instant t),

E(t) : vecteur à k composantes des entrées primaires,

C : matrice de connexion des entrées primaires (ici C = Ix : matrice identité),

M : matrice d'ordre k représentant la structure du LFSR Modulaire.

La fonction de sortie peut être représentée par la matrice Q à k composantes :

$$q(t) = Q \,.\, B(t) \quad \text{avec} \quad Q = (\,0\,,\cdot\,,\cdot\,,\cdot\,,0\,,1\,)$$

Le polynôme caractéristique P_G du MISR est défini comme étant le déterminant $|M\text{-}Ix|$.

Analyse algébrique

Le fonctionnement du MISR peut être analysé selon une approche algébrique. Les variations temporelles des entrées e_i sont représentées par les polynômes $E_i(x)$ définis, pour n mots de k bits compactés, par :

$$E_i(x) = \sum_{j=0}^{n-1} e_i(j)\, x^{n-1-j}$$

Les variations temporelles de la sortie q du circuit et l'état B du MISR sont représentées par les polynômes :

$$Q(x) = \sum_{j=0}^{n-1} b_{k-1}(j)\, x^{n-1-j} \qquad \text{et} \qquad B(x) = \sum_{j=0}^{k-1} b_j(n)\, x^j$$

A l'instant 0, le MISR est initialisé à l'état 0 ($b_j = 0\ \forall j$). Le fonctionnement du circuit correspond alors à la division polynomiale suivante :

$$Q(x) = \frac{E(x) - B(x)}{P_G} \qquad \text{avec} \quad E(x) = \sum_{i=0}^{k-1} E_i(x)\, x^i$$

Cette relation provient de la linéarité du fonctionnement du circuit. La compaction d'un bit b sur l'entrée e_i est algébriquement équivalente à l'entrée du polynôme $b \cdot x^i$ sur e_0. Le théorème de superposition pouvant être employé, on obtient l'expression du polynôme E(x) indiquée ci-dessus. Une preuve complète de la relation peut être trouvée dans [Hass 82].

1.2.2.2. Probabilités de masquage d'une compaction par MISR

Le problème du calcul de la probabilité de masquage d'un MISR a été très récemment abordé dans la littérature. Jusqu'alors, les auteurs s'étaient surtout attachés au calcul de la probabilité de masquage d'un LFSR [Davi 80], [Will 86], [Dami 88], [Xavi 89] et l'on peut en trouver un bon aperçu dans [Leve 90a] et [Ivan 92]. De ces études, on retiendra ici seulement le fait que la probabilité asymptotique de masquage d'un LFSR est égale à 2^{-k}.

Cependant, pour le calcul de la probabilité de masquage d'une compaction par MISR, les résultats des LFSR ne peuvent pas s'appliquer directement.

1.2.2.2.1. Influence entre erreurs

Le comportement d'un MISR n'est pas strictement équivalent à celui d'un LFSR. Entre autres, il existe un problème spécifique aux MISR sur les relations

temporelles entre erreurs arrivant sur les différentes entrées parallèles [Hass 84d]. Ce problème d'influence entre erreurs ("error cancellation") est illustré sur la figure 1.2 . Par exemple, si une erreur arrive sur l'entrée E_{i-1} à l'instant t et qu'une autre erreur arrive au cycle suivant sur l'entrée E_i, ces deux erreurs se masquent mutuellement.

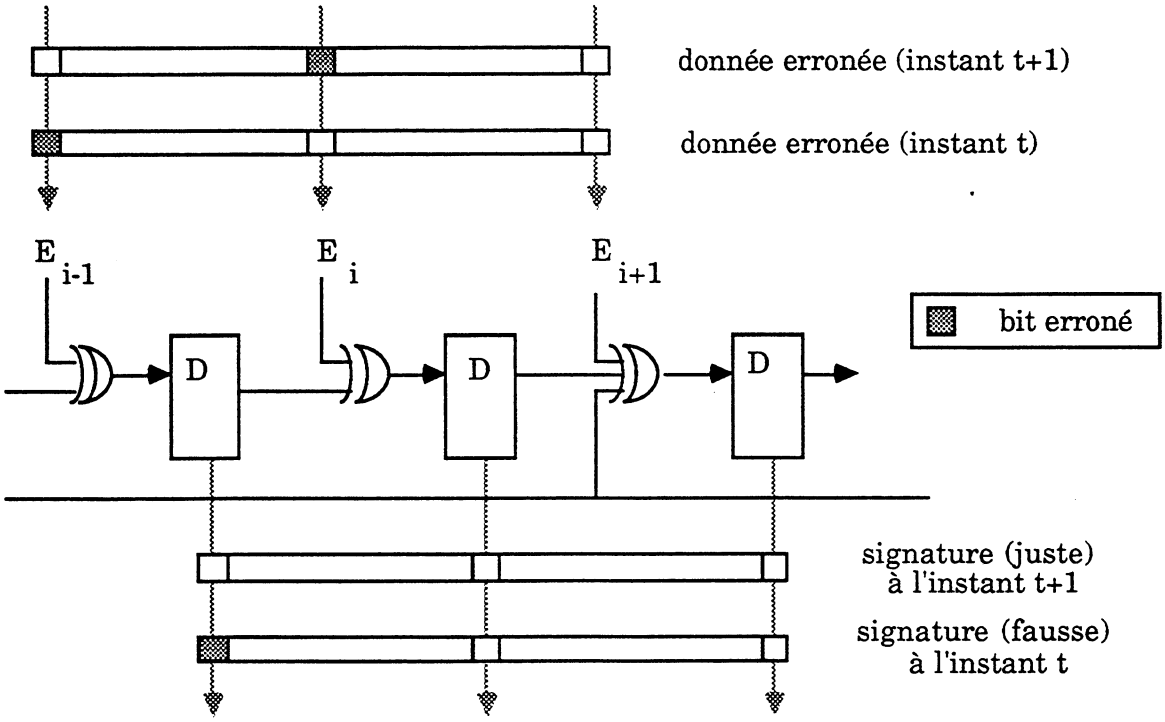


Figure 1.2 : exemple d'influence entre erreurs sur un MISR.

Si l'on considère l'équiprobabilité des vecteurs d'erreurs , la probabilité de masquage due à cet effet est, pour la compaction de n mots de k bits [Hass 84d] :

$$\frac{2^{(n-1)(k-1)} - 1}{2^{nk} - 1} \approx \frac{1}{2^{(n+k)}}$$

où

n est la séquence de la longueur compactée en nombre de mots et

k est la taille du MISR en nombre de bits.

L'utilisation d'une structure équivalente à un LFSR (cf 1.2.2.3) permet de s'affranchir de ce problème d'influence entre les erreurs.

1.2.2.2.2. Modèle d'indépendance des erreurs

Les premières publications traitant explicitement des MISR en considérant le modèle d'indépendance des erreurs sont très récentes [Will 89], [Dami 89]. Ce

modèle est basé sur le fait que les erreurs possibles au sein d'une séquence vont apparaître de manière aléatoire et indépendamment, avec une probabilité fixe p . La distribution des séquences d'erreurs suit alors une loi binomiale. Cela signifie que l'hypothèse est faite qu'il n'y a aucune corrélation entre les erreurs pouvant être compactées dans des mots différents. Dans le cas du test hors ligne, ceci est justifié si les vecteurs d'entrée du circuit sous test sont aléatoires.

Dans ces conditions, la valeur asymptotique 2^{-k} est également atteinte par les MISR avec un polynôme diviseur primitif. Par ailleurs, dans ce cas, cette probabilité asymptotique n'est pas modifiée par une permutation des entrées [Will 89].

Une expression simplifiée de la probabilité de masquage d'un MISR avec un polynôme caractéristique primitif, si l'on suppose l'indépendance des erreurs sur les différentes entrées du MISR, est donnée dans [Dami 89] :

$$Pal \approx \frac{1}{2^k} + \left(1 - \frac{1}{2^k}\right) \prod_{j=1}^{k} |1 - 2p_j|^{n \frac{2^{(k-1)}}{2^k - 1}} - \prod_{j=1}^{k} (1 - p_j)^n \qquad \{P\}$$

Dans le cas de la vérification en ligne d'un flot de contrôle par analyse de signature, l'hypothèse d'indépendance entre erreurs n'est cependant pas toujours justifiable.

Deux cas au moins ne permettent pas l'utilisation du modèle d'erreurs indépendantes :

- suite à une erreur, la séquence des vecteurs d'entrées du MISR est complètement modifiée de manière définitive (cas d'un déséquencement du microprocesseur vérifié, par exemple),
- une même erreur est compactée plusieurs fois (cas où il existe une erreur permanente dans une boucle du graphe du flot de contrôle vérifié et qu'aucun test n'est effectué pendant l'exécution de cette boucle).

Des différentes études menées, il ressort cependant que le polynôme diviseur doit être choisi parmi les polynômes premiers primitifs (ceci implique en particulier que le coefficient de degré 0 soit non nul). Le degré doit être aussi grand que possible, ou du moins établi comme un compromis entre le coût matériel et la probabilité de masquage.

Les caractéristiques dynamiques précises, liées au type de rebouclage, ne peuvent être déterminées que par simulation (transformée en Z ou méthode itérative) du MISR choisi. Il semble toutefois que ces caractéristiques sont meilleures lorsque le polynôme diviseur a beaucoup de coefficients non nuls [Oliv 89]. Un tel choix conduit, dans le cas d'un polynôme câblé, à augmenter le

nombre de portes OU Exclusif à implanter. Un compromis est donc à établir entre le coût du circuit de compaction et la qualité des caractéristiques dynamiques liées au masquage.

1.2.2.3. Structures équivalentes à un LFSR

L'intérêt des structures équivalentes à un LFSR est de supprimer le problème d'influence entre erreurs car, d'un point de vue matériel, ces structures sont plus coûteuses. Pour le calcul de leur probabilité de masquage, il faut alors utiliser les résultats théoriques obtenus pour les LFSRs, qui sur ce plan sont plus simples que les MISR et ont été étudiés depuis plus longtemps.

Par ailleurs, les structures équivalentes à un LFSR permettent un meilleur diagnostic que les MISR. Il existe en effet une information utilisable dans les signatures fausses [McAn 87] : pour une compaction par LFSR ou équivalent, une signature fausse permet de remonter jusqu'au(x) bit(s) erroné(s), alors que dans le cas d'une compaction par MISR, seul un ensemble de couples (bit erroné , instant) peut être déterminé.

Différentes approches ont été utilisées pour obtenir le comportement d'un LFSR avec une structure à entrées parallèles. La plus simple consiste à sérialiser l'information parallèle, puis à l'envoyer sur l'entrée d'un LFSR [Schw 87]. La fréquence de fonctionnement peut cependant poser des problèmes insurmontables.

Une autre approche, utilisée dans [Fent 84], consiste à utiliser des réseaux de OU Exclusifs qui émulent la compaction série.

Une extension systématique et générale de ces approches, nommée "structure universelle", a été présentée dans [Leve 90a]. Il peut être noté que cette structure induit des coûts matériels très importants.

1.2.3. Conclusion sur le choix d'un dispositif de compaction

Le choix d'un dispositif de compaction linéaire comme fonction de compaction ne fait actuellement aucun doute en raison du faible coût matériel des dispositifs associés, et surtout de la faible probabilité de masquage entraîné par ce type de compaction. Pour diminuer cette probabilité de masquage, le polynôme diviseur sera choisi primitif.

Parmi les différentes structures des dispositifs réalisant une division polynomiale, on ne retiendra ici que le MISR à OU Exclusif internes. Cette structure sera celle utilisée dans les méthodes présentées dans les chapitres 2 à 4.

Plus récemment, certains auteurs ont proposé l'utilisation de CA (cellular automata) comme dispositif de compaction et ont montré que ceux-ci avaient, dans ce cas, des propriétés équivalentes aux MISR [Hort 90] .

1.3. Niveau du test : programme ou graphe d'états

On étudiera deux niveaux d'abstraction pour la vérification d'un flot de contrôle. Ces deux niveaux sont abordés succinctement dans cette section en 1.3.1 et 1.3.2, puis ils sont repris en détail dans les sections 1.4 et 1.5. Le premier niveau est le niveau "programme" dans lequel le GFC est à rapprocher de l'organigramme du programme d'application exécuté par un microprocesseur. Le second niveau étudié est le niveau "graphe d'état" qui correspond à la vérification d'un contrôleur de circuit intégré, dans le cas où ce contrôleur est une machine câblée décrite par un graphe d'états fini. Notons que des techniques existent également pour les contrôleurs microprogrammés [Namj 82c], [Srid 82a], [Iyen 85], [Tung 86], mais elles ne seront pas abordées dans ce chapitre, n'ayant pas d'application dans le cadre de cette thèse.

Le principe de base de la vérification du flot de contrôle reste le même aux deux niveaux, à savoir génération d'une signature à partir d'informations extraites du GFC puis comparaison de cette signature avec une référence. La mise en œuvre, par contre, diffère notablement, aussi bien en ce qui concerne les informations compactées, que la manière de stocker les références pour le test.

1.3.1. Test d'un microprocesseur au niveau programme

Pour le test au niveau programme, il existe plusieurs manières d'extraire une signature du flot de contrôle d'un microprocesseur. Mis à part les cas rarissimes où le générateur de signature est intégré directement au processeur, la génération de la signature du GFC est rendue difficile à cause de problèmes d'observabilité du processeur. En effet, la signature doit être générée à partir d'informations dûment identifiées circulant sur les bus du processeur et ceci limite fortement les possibilités si l'on veut rester dans des limites acceptables d'un point de vue coût matériel. On distinguera ici deux types de signatures d'un GFC de programme suivant que les informations utilisées pour générer la signature sont des informations du GFC à proprement parler (signatures dérivées) ou bien des informations rajoutées dans le seul but de générer une signature (signatures imposées).

1.3.1.1. Signatures imposées

Cette première manière de générer une signature du GFC d'un processeur est aussi la plus ancienne et la plus économique. Elle consiste à insérer dans le

programme des informations qui seront utilisées exclusivement pour générer la signature [Yau 80], [Lu 82]. On parlera alors de signature imposée, car les informations utilisées pour générer la signature ne font pas partie du GFC proprement dit, et leur valeur (choisie d'une manière arbitraire) ne joue aucun rôle sur le fonctionnement du système vérifié. Ce mode de vérification permet de vérifier uniquement le séquencement du programme exécuté :

- [Lu 82] vérifie l'organigramme d'un programme par rapport aux structures de référence du langage de haut niveau utilisé,
- [Yau 80] vérifie la correction des chemins grâce à un procédé tout à fait original (fondé sur la décomposition en nombres premiers de la signature) mais particulièrement délicat et coûteux à mettre en œuvre au niveau du test de la signature.

1.3.1.2. Signatures dérivées

La seconde manière de générer une signature du GFC d'un processeur consiste à compacter la suite des instructions exécutées[1]. On parlera alors de signature dérivée car les informations introduites dans la signature sont des informations du GFC proprement dit et que leur valeur (déterminée) influence directement le comportement du système vérifié. Par rapport à une signature imposée, l'utilisation d'une signature dérivée au niveau programme, permet, en plus de la vérification du séquencement, de détecter également les erreurs de bits dans le code du programme exécuté (opérations commandées au processeur). De nombreuses méthodes de vérification d'un flot de contrôle au niveau programme par signature dérivée ont été proposées ces dernières années et les principales d'entre elles seront décrites dans la section 1.4 .

Dans les méthodes à signature dérivée, seules sont vérifiées, en général, la légalité des chemins et les opérations commandées sur les divers noeuds du GFC. La correction des chemins pose un problème pour le stockage des références, comme dans le cas d'une signature imposée [Yau 80]. La vérification, par analyse de signature dérivée, de l'exécution des opérations commandées se heurte d'une part au problème de l'observabilité du processeur pour générer la signature mais également au problème de la génération et du stockage des références.

---

[1] Dans le cas d'un générateur de signature externe pour un processeur disposant d'une file d'instructions avec lecture anticipée, les instructions compactées sont en fait les instructions lues [Delo 91].

Les seules exécutions erronées qui peuvent être détectées par une analyse de signature le sont en fait par effets de bord (par exemple, mauvaise exécution se traduisant par une altération de la structure du GFC).

1.3.2. Test d'un circuit au niveau graphe d'états

Pour le test au niveau graphe d'états, le GFC est à rapprocher du graphe de la machine à états finis décrivant le fonctionnement du contrôleur. La signature est générée à partir de la suite des codes des états parcourus par le contrôleur du circuit (machine câblée). La signature est donc de type dérivé, car l'information utilisée pour générer la signature est l'état du contrôleur, donc détermine en partie le fonctionnement du système vérifié. La méthode, proposée dans [Leve 90a], sera détaillée dans la section 1.5 .

Comme dans le cas d'une signature imposée au niveau programme, seules les erreurs de séquencement peuvent être détectées par l'analyse de signature au niveau graphe d'états. Les erreurs d'opérations nécessitent un dispositif supplémentaire. L'observabilité du système vérifié n'est pas un problème ici puisque cette technique ne peut s'appliquer que dans le cas d'une implantation interne au circuit, mais c'est plutôt le problème du stockage des références qui est critique, car il faut que le volume de matériel rajouté au circuit pour le test soit aussi faible que possible, pour respecter les contraintes liées au coût d'intégration.

1.4. Test au niveau programme

Mise à part la section 1.4.1 qui concerne des généralités sur la structure d'un programme, cette section se rapporte exclusivement au test au niveau programme par signature dérivée.

1.4.1. Représentation d'un programme par un graphe

Un programme possède une structure naturelle de graphe dans laquelle chaque instruction est un noeud et où les cheminements possibles entre ces instructions sont les arcs du graphe. Les informations associées à chaque instruction sont une adresse (numéro d'instruction) et un code opératoire suivi d'une liste de paramètres qui peut être vide. Le code opératoire définit l'opération devant être effectuée sur ce noeud. Pour une bonne partie des instructions, le seul cheminement possible consiste à aller à l'instruction située à l'adresse suivante dans le programme. On dira alors que les instructions sont exécutées linéairement. Les autres instructions sont des instructions de séquencement. Dans l'ensemble des arcs du GFC associé à un programme, on distinguera donc des arcs linéaires et des arcs de séquencement.

Définition 1.8 : un arc linéaire relie deux instructions exécutées en séquence.

Définition 1.9 : un arc de séquencement relie une instruction de séquencement à l'une de ses destinations.

Définition 1.10 : une instruction de séquencement est une instruction qui peut provoquer une rupture de séquence dans le programme. Ces instructions peuvent pour certaines être exécutées de manière conditionnelle ce qui crée une divergence dans la structure du graphe.

Définition 1.11 : une instruction destination est une instruction qui peut être atteinte suite à une rupture de séquence provoquée par une instruction de séquencement. Par la suite, on utilisera le terme "destination" pour désigner une instruction destination.

Du point de vue du séquencement d'un programme, il est possible de regrouper toutes les instructions qui ne sont reliées entre elles que par des arcs linéaires. Ceci permet d'obtenir une représentation réduite de la structure du programme par un graphe orienté dans lequel les blocs linéaires d'instructions sont les noeuds.

Définition 1.12 : un bloc linéaire d'instructions est un sous graphe du GFC ne comportant ni arc de séquencement ni point de jonction. Par la suite, on utilisera le terme "bloc linéaire" pour désigner un bloc linéaire d'instructions.

Il existe différentes sortes de blocs linéaires (Figure 1.3) suivant la nature des instructions marquant les extrémités du bloc :

- un début de bloc linéaire est soit une destination (notée "Dest." sur la figure 1.3), soit une instruction située immédiatement après une instruction de séquencement conditionnelle (notée "BR cond." sur la figure 1.3),
- une fin de bloc linéaire est soit une instruction de séquencement (conditionnelle ou non, notée "Instr. séq" sur la figure 1.3), soit une instruction précédant un point de jonction.

Il existe des cas particuliers de blocs linéaires ne contenant qu'une seule instruction.

Définition 1.13 : un bloc linéaire dégénéré est un bloc linéaire qui contient pour unique instruction une instruction de séquencement.

Il existe quatre types de blocs linéaires différents que l'on notera par la suite A, B, C et D (Figure 1.3).

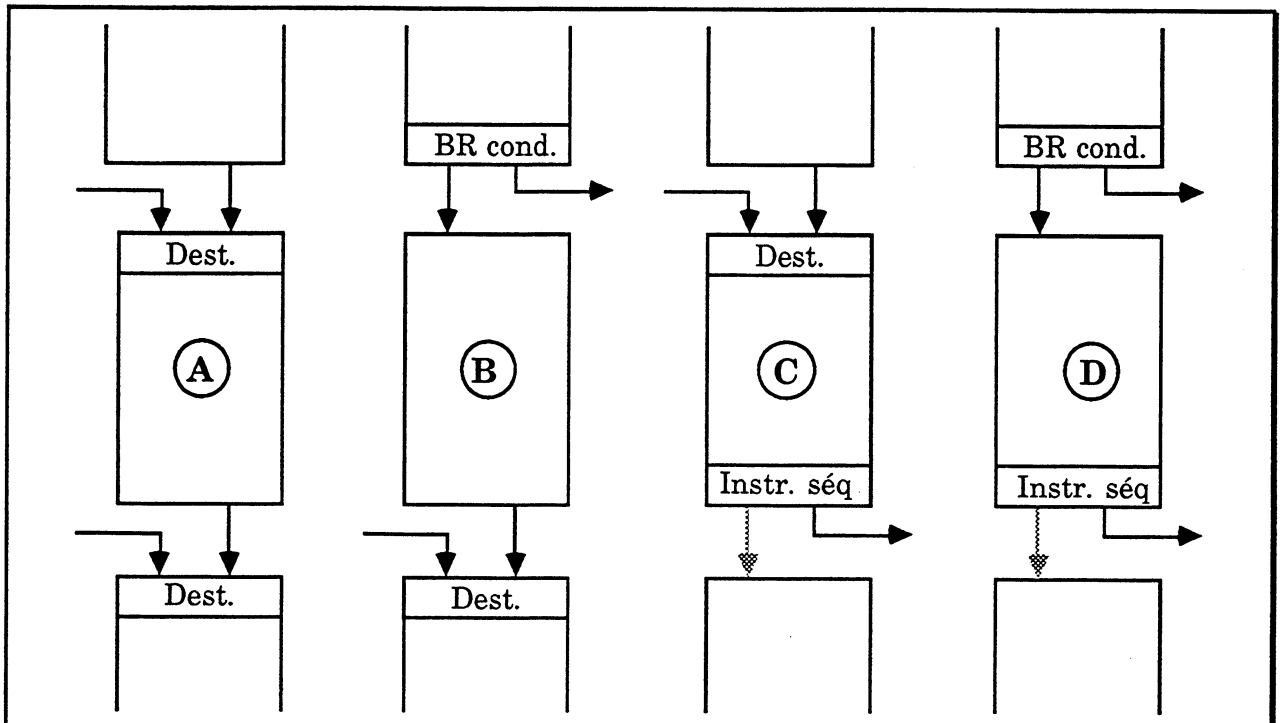
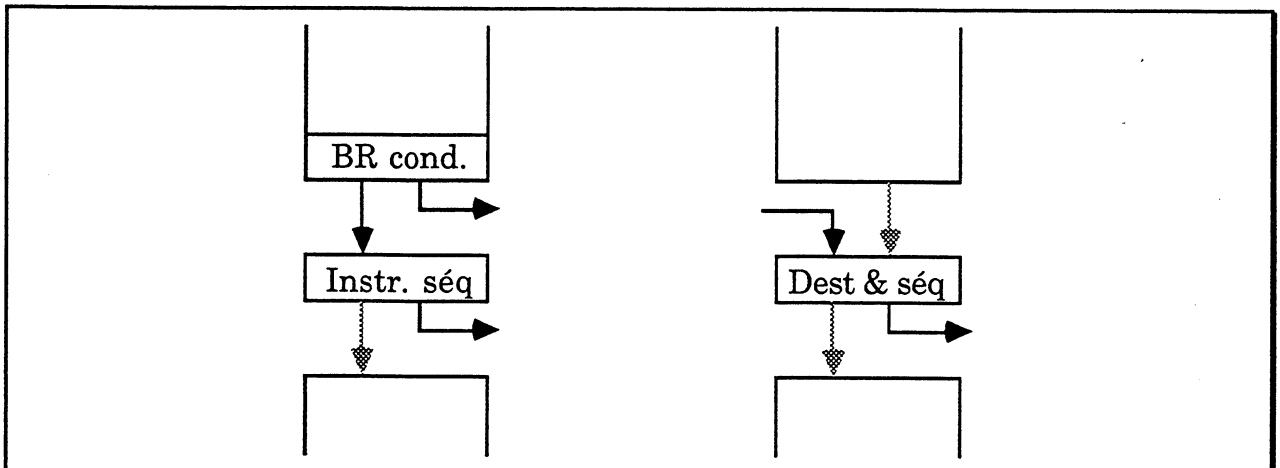


Figure 1.3 : différents types de blocs linéaires.

Il existe deux type de blocs linéaires dégénérés suivant que la seule instruction du bloc est une destination ou non (Figure 1.4).



(a) branchements consécutifs

(b) instruction séquencement et destination

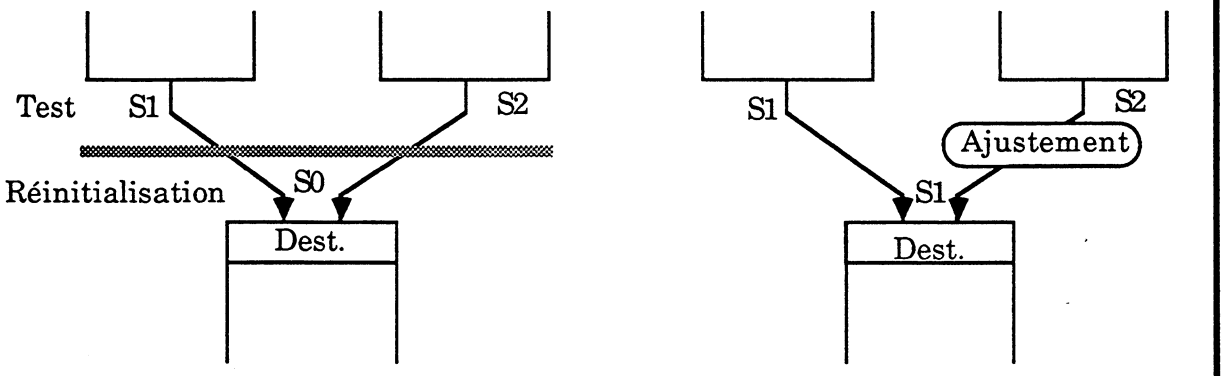
Figure 1.4 : blocs linéaires dégénérés.

1.4.2. Invariance et test de la signature

Le test de la signature a lieu au niveau de certaines instructions du programme par la comparaison d'une valeur de référence précalculée avec la signature obtenue en ligne. Pour permettre le pré-calcul des références, il faut que celles-ci ne dépendent pas du cheminement légal suivi par le processeur pour arriver à une instruction sur laquelle a lieu un test. On dira alors que la signature est invariante aux points de test. Pour satisfaire cette propriété, une condition suffisante mais non nécessaire est que la signature soit invariante au niveau de chaque instruction du programme (invariance des signatures intermédiaires[1]). A de rares exceptions près, tous les schémas d'analyse de signature au niveau programme respectent cette condition pour assurer l'invariance aux points de tests.

Pour assurer l'invariance de toutes les signatures intermédiaires d'un programme, il suffit que les signatures intermédiaires de tous les points de jonction soient invariantes.

Deux méthodes sont utilisées pour assurer l'invariance des signatures intermédiaires aux points de jonction. Elles ont été proposées toutes les deux par Namjoo dans [Namj 82a].



(a) réinitialisation du dispositif de compaction

(b) ajustements

Figure 1.5 : invariance de signature aux points de jonction.

La première méthode consiste à tester la signature puis à réinitialiser le dispositif de compaction avec une valeur fixe à chaque fois que le processeur exécute une instruction correspondant à un point de jonction (Figure 1.5.a).

---

[1] La signature intermédiaire à une instruction est égale au contenu du dispositif de génération de signature après compaction de cette instruction.

La seconde méthode consiste à introduire une correction de signature sur tous les chemins arrivant à un point de jonction, sauf un (Figure 1.5.b). Cette correction est appelée un "ajustement de signature" et la manière la plus courante de procéder est de réaliser le OU Exclusif bit à bit de la signature avec une valeur précalculée. La fonction logique choisie pour réaliser un ajustement n'a pas d'importance à condition que celle-ci soit bijective, c'est-à-dire qu'une signature fausse ne doit pas pouvoir être rendue juste suite à un ajustement.

Un problème particulier se pose pour assurer l'invariance de la signature lorsque le processeur débute le traitement d'une exception (interruption matérielle, trap ...). En effet, la première instruction de la routine d'exception est un point de jonction et il faut donc que la signature y soit invariante. Il n'est cependant pas possible de faire un ajustement ou un test car l'occurrence d'une exception n'est pas prévisible. La solution généralement utilisée consiste à empiler la signature courante puis à réinitialiser le dispositif de compaction lors d'un départ en exception. Lors du retour d'exception, la signature est normalement testée avec une référence précalculée puis la signature du programme interrompu est restaurée.

1.4.3. Localisation et repérage des informations pour le test

Les informations nécessaires à la vérification par analyse de signature du flot de contrôle d'un programme sont de deux types. Des informations doivent assurer l'invariance de la signature (par réinitialisation ou ajustement) et d'autres doivent permettre de tester la signature avec des valeurs de référence.

Définition 1.14 : On appellera singularités les instructions du programme sur lesquelles doit avoir lieu un ajustement, un test ou une initialisation de la signature.

Pour chaque singularité, il faut disposer :
- d'un moyen de repérage dans le programme,
- d'un moyen d'identification du type de la singularité,
- d'une valeur permettant de tester, de réinitialiser ou d'ajuster la signature.

Les valeurs associées aux singularités peuvent être soit :
- implicites (ex : réinitialisation avec une valeur fixe),
- stockées dans le code du programme ; on parlera dans ce cas de méthodes ESM (Embedded Signature Monitoring),
- stockées dans une zone mémoire séparée de celle du programme ; on parlera alors de méthodes DSM (Disjoint Signature Monitoring).

Les méthodes ESM imposent une modification systématique du programme vérifié et une baisse de performance du système car lorsqu'il rencontre une

singularité, le processeur vérifié doit exécuter un ou plusieurs NOP (codes "No OPération) pendant que le moniteur d'analyse de signature traite la singularité. Dans les méthodes DSM au contraire, le programme vérifié n'est pas modifié et le système peut ne pas être ralenti si les références [1] sont de plus stockées dans une mémoire physiquement séparée de celle contenant les instructions du programme, comme indiqué sur la figure 1.6 .

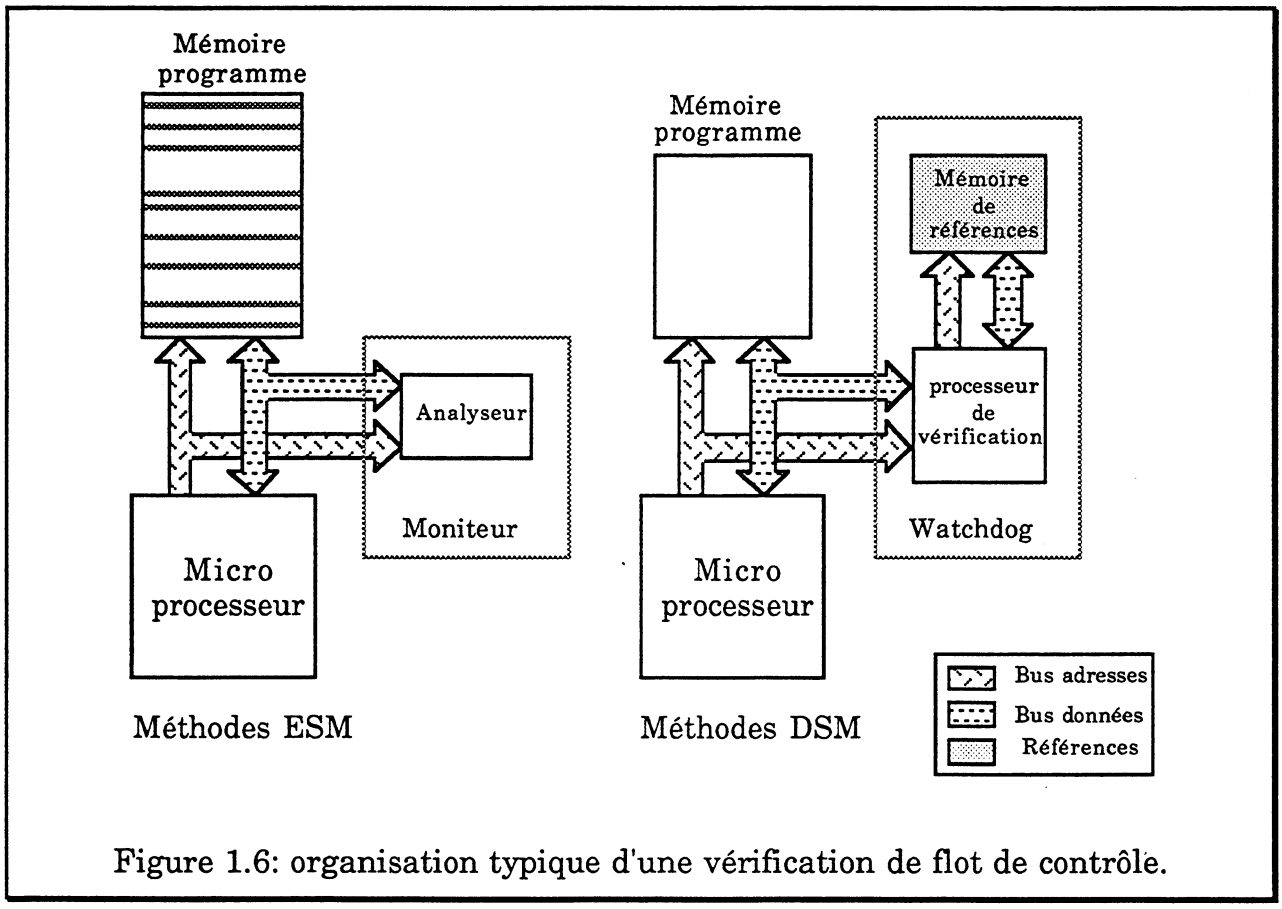


Figure 1.6: organisation typique d'une vérification de flot de contrôle.

Le repérage des singularités d'un programme peut se faire de différentes manières. Les solutions trouvées dans la littérature sont le décodage des instructions, le repérage relatif et le repérage par mémoire étiquette.

- Décodage des instructions : [Shen 83], [Eife 84], [Wilk 87], [Sosn 88], [Saxe 89], [Made 91a], dans ce cas, les singularités doivent correspondre à un type d'instruction particulier et le décodage des instructions indique la présence et le type de la singularité. Ce mode de repérage est le plus utilisé notamment pour les méthodes ESM. Dans ce cas, les singularités sont des instructions dédiées rajoutées

---

[1] Par abus de langage, on appellera "références" les valeurs associées aux singularités, quel que soit leur type.

et les valeurs associées aux singularités sont les paramètres de ces instructions. Des instructions non rajoutées peuvent également être des singularités repérées par décodage, par exemple une instruction de séquencement repère toujours une fin de bloc linéaire.

- Repérage relatif : [Shen 83], [Namj 83], [Saxe 89], dans ce cas, une singularité est repérée par le nombre d'instructions qui la sépare de la singularité précédente. Un type de repérage relatif particulier concerne les extrémités de blocs linéaires : un début de bloc linéaire est toujours situé immédiatement (donc une instruction) après une fin de bloc linéaire. Mis à part ce cas particulier fréquent, ce type de repérage n'est quasiment pas utilisé.

- Mémoire étiquette externe : [Namj 82a], [Delo 90], [Wilk 88], [Upad 91], [Made 91a], dans ce cas, une mémoire de quelques bits est ajoutée en parallèle de la mémoire contenant le code du programme vérifié. Cette mémoire est donc lue en même temps que la mémoire instruction et le champ de bit supplémentaire sert à coder la présence et le type d'une singularité.

Remarques :

-> Un repérage des singularités par mémoire étiquette peut remplacer un repérage par décodage d'instructions dans les cas suivants :

- méthodes DSM car il est impossible d'insérer une instruction pour marquer la présence d'une singularité,
- processeur dont le format d'instruction est complexe, donc coûteux à décoder pour isoler les codes opératoires.

-> Il est possible pour un même schéma d'analyse de signature de repérer différents types de singularités de différentes manières comme en témoignent les références citées sur plusieurs méthodes de repérage.

1.4.4. Méthodes ESM

1.4.4.1. Méthodes ESM sans ajustements

Les méthodes ESM qui n'utilisent pas d'ajustement pour assurer l'invariance de la signature ne sont pas très nombreuses. On citera ici les principales, c'est-à-dire PSA (Path Signature Analysis) de base (PSAb) [Namj 82a], SIS (Signatured Instruction Stream) [Shen 83], [Schu 87] et la méthode présentée dans [Saxe 89], [Saxe 90]. PSA et SIS ont été les premières méthodes connues de vérification de flot de contrôle par signature dérivée, mais leur intérêt n'est pas seulement historique. L'originalité de la méthode de [Saxe 89] vient de l'utilisation d'une fonction de compaction monotone qui permet de réduire la latence de détection des erreurs de séquencement.

Sur le principe, ces méthodes sont très proches : elles consistent à affecter une signature de référence par bloc linéaire d'instruction. Pour PSAb et SIS, la signature est réinitialisée avec une valeur fixe en début de bloc linéaire et elle est comparée avec la référence du bloc en fin de bloc. Dans la méthode [Saxe 89] par contre, la signature est réinitialisée en début de bloc avec la signature de référence du bloc et elle est comparée à une valeur fixe en fin de bloc.

1.4.4.1.1. Méthode PSA de base [Namj 82a]

PSAb utilise une mémoire étiquette externe de deux bits pour localiser les débuts et les fins de blocs linéaires. Les signatures de référence se situent en tout début de bloc et sont donc localisées par l'identificateur de début de bloc (Figure 1.7). Lorsque le processeur arrive à la fin du bloc, repérée par un identificateur particulier, la signature est testée avec la référence lue en début de bloc, puis elle est réinitialisée.

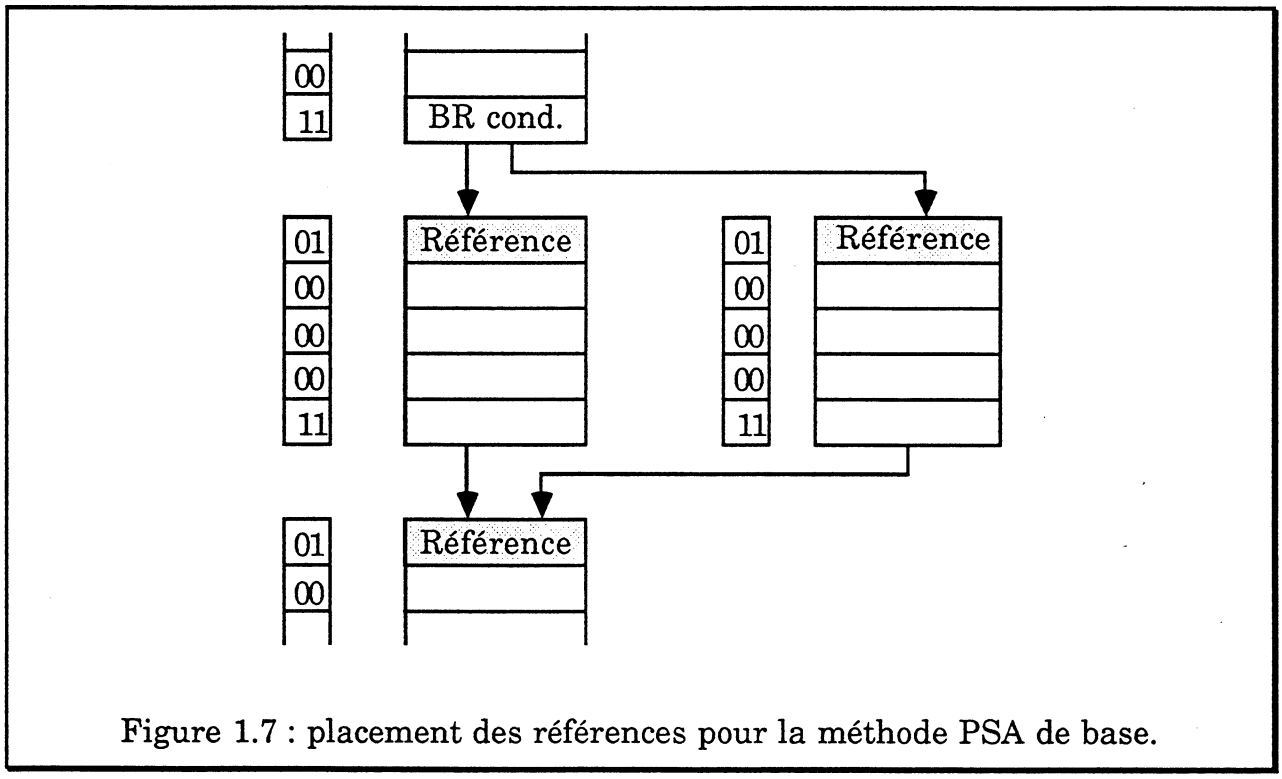


Figure 1.7 : placement des références pour la méthode PSA de base.

La présence des ces bits externes repérant les extrémités de blocs permet de ne stocker qu'un seul mot de référence par bloc linéaire et de s'affranchir d'un décodage des instructions du processeur. Mais le principal intérêt de ce type de repérage pour les extrémités de blocs est de permettre une détection des ruptures de séquences inopinées. En effet, les branchements ne sont possibles que sur une instruction de fin de bloc. Tout branchement en dehors de ces instructions est une

rupture de séquence inopinée qui sera détectée par un test de la séquentialité des adresses programme à l'intérieur des blocs linéaires.

Par ailleurs, la destination d'un branchement doit être un début de bloc et il est donc également possible de détecter les branchements à une adresse erronée. En particulier, un saut à une adresse située hors du code est détecté. Pour les branchements erronés mais dont la destination reste à l'intérieur du code, un masquage peut se produire dans le cas où la destination erronée est un début de bloc.

Il faut cependant remarquer qu'une erreur mémoire pouvant entraîner un branchement erroné (erreur sur le code opératoire d'une instruction de branchement ou sur le paramètre pour le calcul d'une adresse destination) sera détectée par le test de la signature au niveau de la fin du bloc linéaire et non par la vérification de la mémoire étiquette au niveau la destination (détection avant l'exécution du branchement).

1.4.4.1.2. Méthode SIS [Shen 83]

A la différence de PSAb, SIS n'utilise pas de mémoire étiquette externe et le repérage des extrémités de blocs se fait d'une manière mixte, par décodage des instructions pour les fins de blocs et par repérage relatif (1 instruction par rapport à une fin de bloc) pour les débuts de blocs. Les signatures de référence sont stockées comme paramètres des instructions identifiant les fins de bloc linéaire. Pour les blocs se terminant sur une instruction de séquencement, une réduction du coût mémoire est obtenue par l'emploi d'une technique appelée BAH (Branch Address Hashing, figure 1.8).

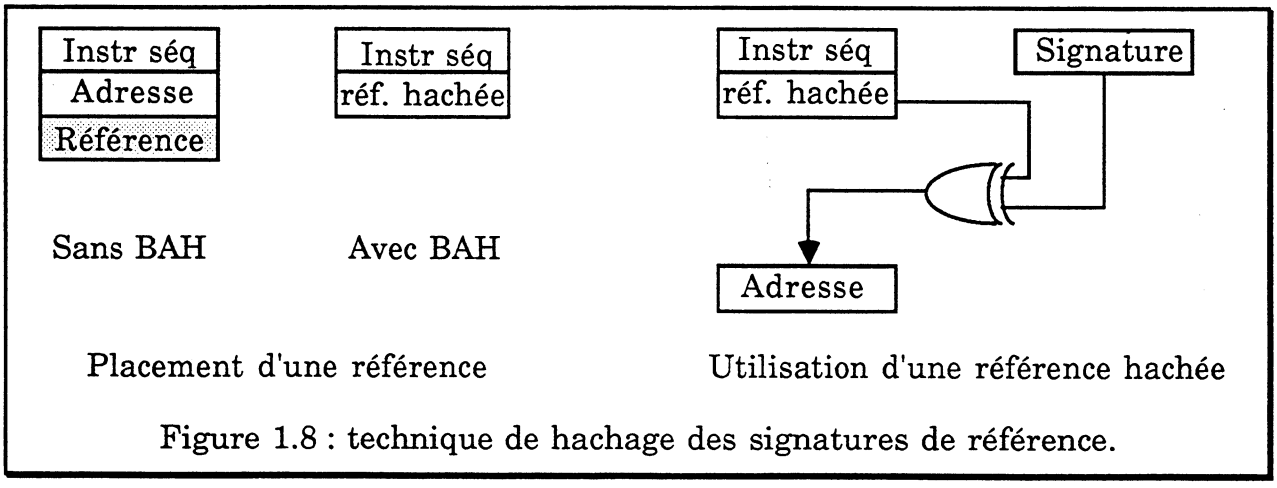


Figure 1.8 : technique de hachage des signatures de référence.

Cette technique consiste à remplacer l'adresse de destination de l'instruction de séquencement par le OU Exclusif entre cette adresse et la signature de référence (hachage). Lors de l'exécution du programme, un hachage inverse entre la

signature courante et la référence hachée permet de retrouver l'adresse de destination de l'instruction de séquencement (Figure 1.8). Ainsi, une erreur de signature se traduit en fait par une erreur sur le séquencement du processeur. La technique BAH ne s'applique qu'aux blocs se terminant par une instruction de séquencement. Pour les blocs se terminant sur un point de jonction, il faut utiliser une instruction dédiée pour stocker la référence et marquer la fin de bloc (Figure 1.9).

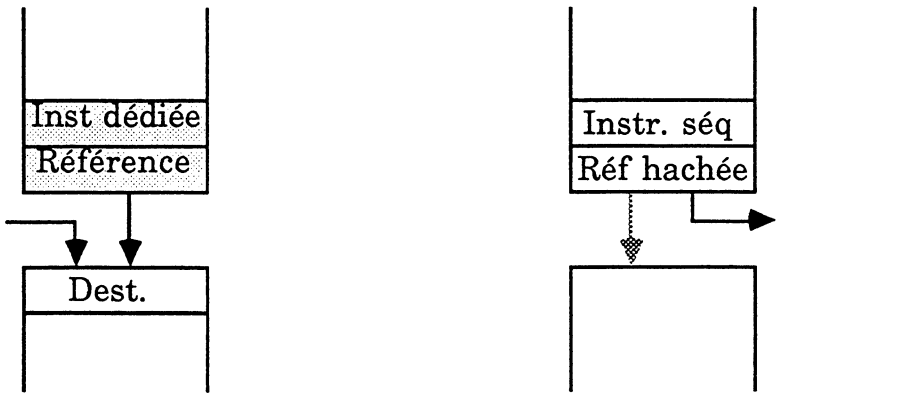


Figure 1.9 : placement des références pour la méthode SIS.

Avec cette méthode de repérage, les retours de sous-programmes constituent une exception. En effet, la fin d'un sous-programme est un bloc linéaire terminé par une instruction de séquencement qui ne possède pas d'adresse destination. Il n'est donc pas possible de hacher la signature de référence. Il faut alors mettre une instruction dédiée pour stocker la référence juste avant l'instruction de retour. Dans ce cas, la première instruction du bloc située après le départ en sous-programme est l'instruction de retour de sous-programme. En fait, dans SIS, la valeur de test est placée immédiatement après l'instruction de retour de sous-programme et avec la recherche anticipée d'instruction du processeur (MC68000 dans ce cas) cette valeur est quand même lue et identifiée comme telle par le moniteur grâce au décodage de l'instruction de retour. Ceci ne fonctionne que pour les processeurs disposant d'un mécanisme de lecture anticipée des instructions.

La technique BAH permet une réduction importante du coût mémoire des références (50% d'après [Shen 83]) mais en contrepartie elle induit un masquage d'erreur très important. En effet, une erreur de signature provoquant une erreur de séquencement à une adresse qui correspond à un début de bloc dans le programme est indétectable. Cette probabilité de masquage est d'autant plus forte que la taille du

programme est importante. Dans le cas de gros programmes, ce masquage peut facilement atteindre des valeurs de l'ordre de 15%. Wilken obtient un résultat théorique similaire (6 à 16%) en utilisant un modèle de Markov [Wilk 87]. Dans le cas de petits programmes par contre, la probabilité de sortir du code est la plus forte et c'est pourquoi il est proposé avec SIS une vérification des bornes du programme, technique qui à elle seule est une méthode de test en ligne très efficace.

Un autre point critiquable de la technique BAH utilisée dans SIS est l'allongement du temps de lecture des instructions. En effet, il faut ajouter le temps nécessaire au hachage inverse (traversée d'un boîtier + fonction logique) au temps de réaction de la mémoire. Ceci entraîne donc une baisse globale des performances du système.

L'absence de mémoire étiquette dans SIS, par rapport à PSAb, empêche une détection instantanée des ruptures de séquence inopinées et des branchements erronés et le fait de réinitialiser la signature sur rupture de séquence peut introduire un masquage. Deux cas de rupture de séquence inopinée doivent être distingués :

- erreur mémoire se traduisant par la lecture d'une instruction de séquencement : dans ce cas la signature est réinitialisée car l'instruction est décodée par le moniteur, et le masquage dépend donc de la probabilité que la destination de ce branchement intempestif soit le début d'un bloc (6 à 16%),
- erreur interne au processeur : dans ce cas, la signature n'est pas réinitialisée et le masquage dépend de la probabilité que la destination du processeur ait une signature intermédiaire théorique égale au contenu du dispositif de compaction à cet instant. Cette probabilité dépend de la répartition des signatures intermédiaires sur le code (corrélation des signatures intermédiaires).

Le cas d'un branchement erroné est identique à celui d'un branchement inopiné sur erreur mémoire.

1.4.4.1.3. Utilisation d'une fonction de compaction monotone [Saxe 89]

L'originalité de la méthode présentée dans [Saxe 89] par rapport à PSAb ou SIS consiste à utiliser une fonction de compaction des instructions telle que la suite des signatures intermédiaires soit monotone, en l'occurrence décroissante. La fonction choisie par Saxena n'est autre que la soustraction. La signature de référence d'un bloc est stockée en début de bloc et elle est égale à la somme des codes du bloc. Dans ce cas, pour assurer la monotonie des signatures intermédiaires, il est impératif que la taille des signatures soit plus élevée que la taille des instructions. Saxena utilise des signatures de longueur égale au double des instructions ce qui entraîne

un coût de stockage des références important ainsi qu'un générateur de signature plus coûteux en matériel.

Le fonctionnement de l'analyseur consiste à initialiser la signature en début de bloc avec la référence placée à cet endroit, puis à chaque lecture d'une instruction, le code de l'instruction est retranché à la signature. La signature en fin de bloc doit normalement être nulle et si tel n'est pas le cas, une erreur est détectée. De même, si la signature devient inférieure ou égale à zéro avant la fin d'un bloc une erreur est détectée. Avec cette technique, la latence de détection des erreurs de séquencement est considérablement réduite en particulier si le processeur sort de la zone contenant le code. De même, la détection des branchements inopinés (sur erreur mémoire) est possible car une instruction de séquencement, toujours située en fin de bloc, doit correspondre à une signature nulle. Les branchements erronés sont détectés avec la même efficacité et avec les mêmes cas de masquage que dans PSAb.

Dans [Saxe 89] le repérage des extrémités de blocs est fait par décodage d'instructions : les débuts de blocs sont marqués par une instruction dédiée permettant le stockage des références, et les fins de blocs sont repérées par le décodage des instructions de séquencement ou relativement à un début de bloc. Ceci porte le nombre de mots introduits dans le code à trois fois celui de PSAb (un mot d'instruction et deux mots de checksum par bloc linéaire) pour des services tout à fait comparables, mais sans recours à une mémoire d'étiquette, et sans la nécessiter de tester la séquentialité des adresses programmes pour détecter les branchements inopinés. Le coût matériel du moniteur est donc largement inférieur à celui de PSAb.

1.4.4.1.4. Remarques sur les méthodes sans ajustements

a/ Coût mémoire et latence de détection

L'avantage de ces méthodes réside dans une latence de détection faible, pour un coût mémoire fixe, et dans la très grande facilité de génération des références. Le coût mémoire est relativement important par rapport aux méthodes avec ajustements car, pour ces dernières, la latence de détection n'est pas fixe (elle est en général beaucoup plus élevé). Cependant, pour une latence comparable, le coût mémoire est à peu près identique dans les deux cas, voire inférieur dans le cas de SIS [Wilk 87].

b/ Corrélation des signatures intermédiaires

L'invariance de la signature est obtenue par réinitialisation du dispositif de compaction ce qui a pour effet indésirable d'entraîner une très forte corrélation sur les valeurs des signatures intermédiaires.

Comme dans toutes les méthodes ESM, une rupture de séquence inopinée ou un branchement erroné entre deux instructions ayant même signature intermédiaire n'entraînent pas d'erreur de signature et ne peuvent donc être détectés à travers la signature. Ce masquage est d'autant plus important que les signatures intermédiaires sont fortement corrélées [Wilk 87]. Pour supprimer cette corrélation, dans SIS et PSAb, il est possible de réinitialiser la signature avec une valeur différente pour chaque début de bloc, mais alors il faut stocker les valeurs de réinitialisation ce qui augmente considérablement le coût mémoire. On peut également réinitialiser avec l'adresse du début du bloc [Made 91a] mais dans ce cas les signatures de référence dépendent de l'adresse de chargement du programme.

Dans tous les cas, le problème n'est que partiellement résolu en raison de l'équivalence des points de jonction. En effet, si la signature est réinitialisée lors d'un branchement erroné sur un point de jonction, la valeur de réinitialisation affectée à ce point de jonction sera utilisée. La signature du bloc linéaire suivant sera donc correcte aussi bien si la valeur de réinitialisation est particulière au point de jonction que si elle est identique pour tous les points de jonction. C'est donc le fait de réinitialiser la signature et non pas la valeur de réinitialisation qui entraîne une possibilité de masquage dans certains cas.

Avec un dispositif détectant les erreurs de séquencement, comme c'est le cas pour PSAb, les ruptures de séquence inopinées et les branchements erronés dont la destination n'est pas un début de bloc sont détectés par ce mécanisme et non pas à travers la signature. La corrélation des valeurs des signatures intermédiaires n'a donc pas d'importance.

Dans tous les cas, la détection des branchements erronés dont la destination est un début de bloc ne peut pas être assurée par une méthode ESM sans ajustements à cause de la réinitialisation de la signature aux points de jonction et ce, quelle que soit la valeur de réinitialisation.

c/ Réinitialisation de la signature

La réinitialisation du dispositif de compaction à chaque branchement du processeur a toutefois l'avantage de permettre une génération de signature déterministe aisée quel que soit le pipeline de traitement des instructions dans le processeur (cf 2.6.2).

1.4.4.2. Méthodes ESM avec ajustements

En section 1.4.2, il a été mentionné que des ajustements devaient être placés sur tous les chemins arrivant à un point de jonction, sauf un, pour que la signature intermédiaire en ce point soit invariante, ce qui suffit à assurer l'invariance de toutes les signatures intermédiaires du programme. Or il existe deux manières pour un processeur d'arriver à un point de jonction. Soit il y arrive en séquence, soit il y arrive suite à une rupture de séquence provoquée par une instruction de séquencement. On voit donc qu'il existe logiquement deux manières d'assurer l'invariance par ajustement à un point de jonction (Figure 1.10). La première méthode consiste à ajuster la signature sur tous les arcs de séquencement arrivant à ce point de jonction. On parle alors d'ajustement sur arcs (sous-entendu "arcs de séquencement"). Le seul chemin ne comportant pas d'ajustement est dans ce cas l'arrivée séquentielle sur le point de jonction. La deuxième méthode consiste à ajuster la signature de l'arc linéaire arrivant à un point de jonction. On parle alors d'ajustement sur noeuds, car l'instruction sur lequel doit avoir lieu l'ajustement appartient au noeud (bloc linéaire) précédant le point de jonction. Si le point de jonction peut être atteint par plusieurs instructions de séquencement, il faut également placer des ajustements dans tous les noeuds contenant ces instructions de séquencement, sauf un.

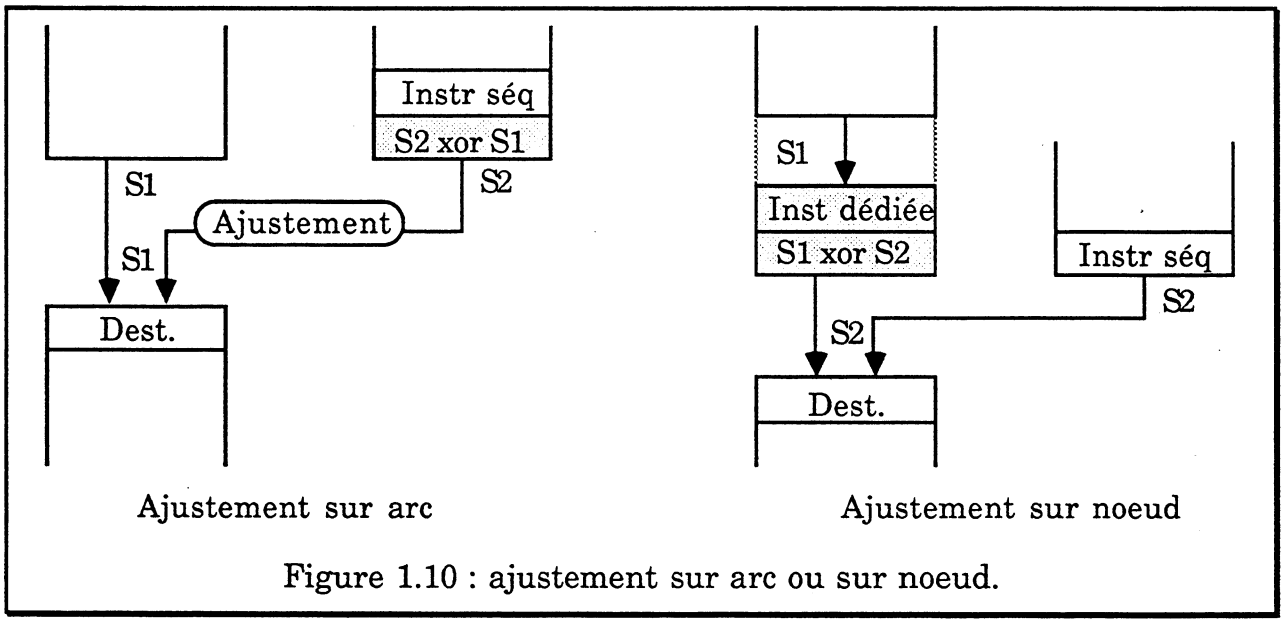


Figure 1.10 : ajustement sur arc ou sur noeud.

Dans les méthodes avec ajustements, une signature n'est pas affectée à un bloc linéaire d'instructions mais à un chemin d'écoulement du programme et de ce fait, la signature contient une information sur le séquencement du programme, ce qui n'est pas le cas pour une signature réinitialisée à chaque point de jonction. L'idée

sous-jacente des ajustements est de rendre tous les chemins d'écoulement du programme équivalents au niveau de leur signature.

Dans les méthodes avec ajustements, il faut insérer dans le programme, outre les valeurs d'ajustement, des singularités pour le test de la signature. Le nombre et l'emplacement de ces singularités déterminent la latence de détection qui peut donc être variable, contrairement aux méthodes sans ajustements. Un compromis peut donc être réalisé entre coût mémoire et latence de détection pour les méthodes avec ajustements.

Le placement des ajustements est un problème non trivial qui doit prendre en compte plusieurs critères tels que coût mémoire, coût en performance, complexité de l'algorithme de placement et facilité de calcul des signatures intermédiaires en tout point du programme.

Il existe un nombre minimal théorique d'ajustements à introduire pour assurer l'invariance de la signature en tout point d'un programme. Ce nombre dépend évidemment du nombre de chemins dans le programme.

S'il existe M chemins maximaux dans un programme comportant un seul point d'entrée et un seul point de sortie, alors il faut au minimum M-1 ajustements pour que tous les chemins du programme génèrent la même signature [Jay 86].

Dans un programme comportant un seul point d'entrée, un seul point de sortie, ne comportant pas d'états ayant plus de deux successeurs et comportant N branchements conditionnels, le nombre de chemins maximaux dans ce programme est égal à N+1 [Wilk 88], et il faut donc un minimum de N ajustements pour assurer l'invariance des signatures intermédiaires de ce programme, indépendamment du fait que les ajustements soient faits sur noeuds ou sur arcs. Ce nombre minimal d'ajustements à introduire est cependant difficile à obtenir et il dépend de l'algorithme de placement utilisé.

1.4.4.2.1. Méthodes ESM avec ajustements sur arcs

Les ajustements sur arcs ont été introduits par K. Wilken [Wilk 87]. L'intérêt de ce mode de placement vient du fait qu'il n'est pas nécessaire de rajouter des informations pour le repérage (localisation et type) des singularités liées à l'invariance de la signature car le décodage des instructions de séquencement y suffit. Ceci permet donc l'introduction d'un seul mot mémoire par ajustement, les valeurs d'ajustement devant alors être considérées comme des paramètres des instructions de séquencement. Le nombre de paramètres des instructions de séquencement doit donc être augmenté. Cette contrainte peut être facilement prise

en compte dans le cas d'une réalisation interne au processeur, mais dans le cas contraire, il faut utiliser un subterfuge.

K. Wilken propose dans [Wilk 87] de placer la valeur d'ajustement immédiatement après le dernier mot de l'instruction de séquencement. Avec le prefetch du processeur (MC68000 dans ce cas) la valeur d'ajustement est quand même lue par le processeur et donc par le moniteur qui l'identifie comme telle grâce au décodage des instructions de séquencement. Le moniteur doit alors fournir le code d'un NOP au processeur, à la place de la valeur d'ajustement, ceci pour éviter qu'en cas de branchement conditionnel non effectué la valeur d'ajustement soit exécutée comme une instruction par le processeur. En cas de branchement conditionnel non effectué, l'ajustement ne doit pas avoir lieu et il faut donc que le moniteur soit capable de détecter les ruptures de séquence du processeur.

Avec ce type de fonctionnement des ajustements, le coût en performance est très faible car le processeur exécute un NOP uniquement dans le cas d'un branchement conditionnel non pris. Dans tous les autres cas d'instructions de séquencement, le processeur n'est pas ralenti.

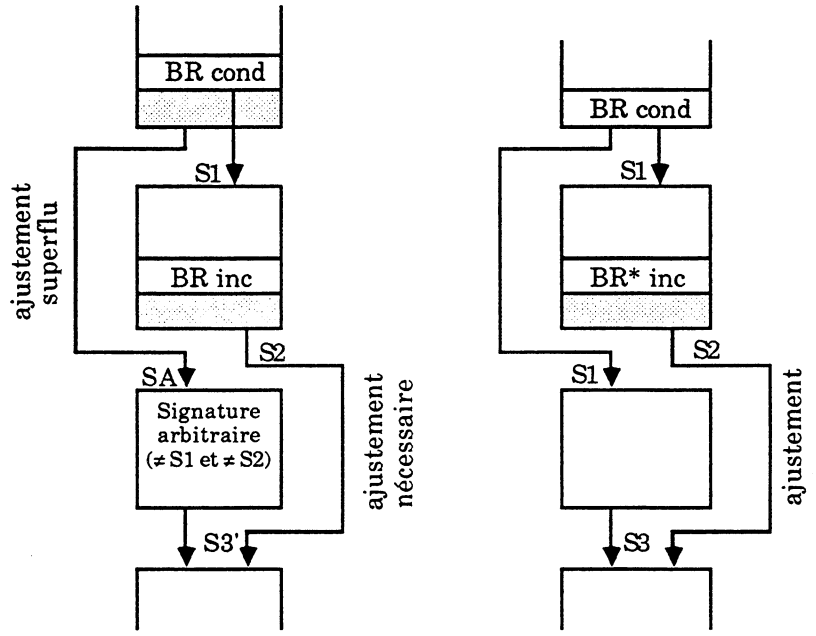
Le coût mémoire de la méthode est également très faible car un ajustement ne prend qu'un seul mot mémoire.

Dans la méthode de base proposée dans [Wilk 87], toutes les instructions de séquencement reçoivent un ajustement (Figure 1.11.a). Le nombre d'ajustements introduits dans le programme est donc assez loin du nombre minimal mais la génération des signatures intermédiaires est très simple car de nombreuses signatures sont arbitraires.

Pour diminuer le coût mémoire, Wilken propose de supprimer les ajustements des instructions de séquencement dont la destination n'est pas un point de jonction (Figure 1.11.b). Cette manière de placer les ajustements complique sérieusement le calcul des signatures intermédiaires et ne permet pas toujours d'atteindre le nombre minimal d'ajustements à inclure pour assurer l'invariance de la signature sur l'ensemble d'un programme (cf 1.4.4.2.4). Par ailleurs, cette solution entraîne une plus forte corrélation des signatures intermédiaires à cause des branchements conditionnels sans ajustement (deux instructions ont S1 comme signature intermédiaire sur la figure 1.11.b).

Cette solution est également plus difficile à implanter car il n'est pas possible pour un moniteur indépendant du processeur de faire une discrimination sur les instructions de séquencement. Ceci nécessite en effet l'utilisation de deux codes distincts pour chaque instruction de séquencement suivant que l'arc de séquencement de cette instruction comporte ou non un ajustement (BR et BR* sur la

figure 1.11.b). Ceci n'est possible que dans le cas d'un processeur spécifique dont le jeu d'instruction possède de nombreux codes non affectés. La seule solution envisageable pour un moniteur autonome consiste à utiliser un bit d'étiquette pour discriminer les instructions de séquencement ce qui, au niveau du coût d'implantation global, fait plus que perdre le bénéfice ainsi gagné sur le nombre d'ajustements, à moins de le placer d'une manière astucieuse comme dans CSM [Wilk88] (cf 1.4.4.2.5).



(a) ajustements sur tous les arcs

(b) ajustements sur les arcs conduisant à un point de jonction

(structure IF THEN ELSE)

Figure 1.11 : placement des ajustements sur arcs [Wilk 87].

L'introduction de signatures de référence sur arcs nécessite de définir un type d'instruction de séquencement particulier sur lesquelles doit avoir lieu un test (par exemple les retours de sous-programmes dans [Wilk 87]). Il reste bien sûr possible de rajouter des références sur certains noeuds du programme pour diminuer la latence de détection mais dans ce cas et si le repérage est fait par instruction dédiée (ce qui est le plus logique), le coût mémoire d'un test est de deux mots au lieu d'un seul.

1.4.4.2.2. Méthodes ESM avec ajustements sur noeuds

Dans ce cas, les ajustements sont effectués dans un bloc linéaire, en général avant un point de jonction. Cette méthode est la plus répandue car sa mise en œuvre est relativement aisée aussi bien dans le cas d'un moniteur intégré au processeur que dans le cas d'un moniteur externe [Namj 82a], [Srid 82], [Jay 86], [Sosn 88], [Leve 90c], [Delo 90], [Mich 92]. Les deux types de singularités (valeurs d'ajustement et signatures de référence) sont introduites dans certains noeuds du programme. Pour le repérage de ces singularités, la plupart des schémas proposés utilisent des instructions dédiées. Par rapport à un ajustement sur arc, le coût mémoire d'un ajustement sur noeud est donc toujours supérieur car il faut deux mots (une instruction et une valeur d'ajustement) alors qu'un seul suffit pour les ajustements sur arcs. Le coût en performances est également supérieur a priori car tous les ajustements impliquent l'exécution de NOPs alors que seuls les branchements conditionnels non effectués induisent un NOP dans le cas des ajustements sur arcs de la méthode [Wilk 87].

Pour réduire le coût en mémoire et donc le coût en performance, il est possible d'utiliser une mémoire externe d'étiquette pour le repérage des singularités [Namj 82a]. Cette technique est par ailleurs indispensable pour implanter un moniteur séparé dans le cas d'un processeur ne disposant pas de codes opératoires inutilisés (ou inutiles) dans son jeu d'instruction. Cette technique permet également de s'affranchir du décodage des instructions du processeur et on peut donc l'utiliser avantageusement en cas de format d'instruction trop coûteux à décoder.

1.4.4.2.3. Traitement des sous-programmes

Les sous-programmes constituent un cas particulier dans les méthodes avec ajustements. En effet, quel que soit le schéma d'ajustement retenu, en cas de départ en sous-programme il faut ajuster la signature avant le début du sous-programme, car un début de sous-programme est un point de jonction, et il faut que sa signature soit invariante. En théorie, si un sous-programme est appelé à M endroits différents dans le programme, alors il faut M-1 ajustements pour assurer l'invariance de ce sous-programme. Un appel de sous-programme peut donc être théoriquement traité comme n'importe quelle instruction de séquencement vis à vis des ajustements [Sosn 88].

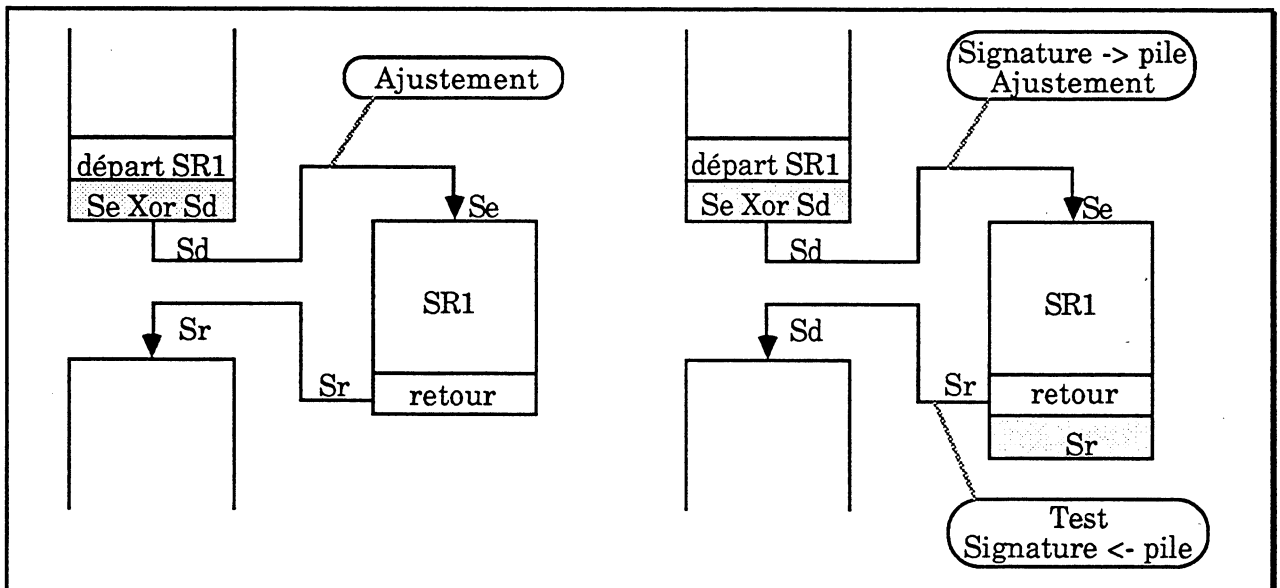
En fait, le problème des sous-programmes vient de l'instruction de retour de sous-programme. En effet, il s'agit d'une instruction de séquencement dont le nombre de destinations possibles est en général supérieur à deux ce qui l'assimile à un branchement à destinations multiples. Comme il n'est pas possible d'ajuster sur la valeur intermédiaire de la destination, toutes les destinations doivent avoir la

même signature intermédiaire. Ceci est donc néfaste pour la décorrélation des valeurs des signatures intermédiaires. Par ailleurs, la signature du programme après un appel de sous-programme dépend de la signature de ce sous-programme, ce qui pose des problèmes pour un calcul modulaire des références d'un programme ou dans le cas de sous-programmes récursifs. Cette manière de faire a cependant l'avantage de présenter un faible coût de réalisation (Figure 1.12.a).

Technique de décorrélation maximale des signatures intermédiaires

Wilken propose dans [Wilk 87] d'empiler la signature au départ en sous-programme, puis d'ajuster la signature sur la valeur d'entrée du sous-programme avec une valeur d'ajustement stockée comme les autres c'est-à-dire immédiatement après le dernier mot de l'instruction de départ en sous-programme (Figure 1.12.b).

Au retour du sous-programme, la signature est testée avec une référence stockée immédiatement après l'instruction de retour, puis la signature est dépilée. Le moniteur doit alors fournir un NOP au processeur à la place de la première instruction exécutée après le retour de sous-programme car il s'agit de la valeur d'ajustement d'entrée de sous-programme, placée immédiatement après l'instruction de départ.



(a) technique à coût minimal [Sosn 88]

(b) décorrélation des signatures de retour [Wilk 87]

Figure 1.12 : invariance des signatures de sous-programmes.

Une méthode similaire peut être implantée avec des ajustements sur noeuds mais beaucoup moins aisément car il faut disposer de deux codes instructions (ou

étiquettes) supplémentaires permettant d'identifier les ajustements d'entrée et les tests de retour qui doivent respectivement empiler et dépiler la signature.

Avec cette technique, les retours de sous-programmes constituent les seuls points de test obligatoires de la signature. La latence de détection minimale est donc en moyenne égale à la moitié du temps moyen entre l'exécution de deux retours de sous-programmes [Wilk 87]. L'avantage de cette technique est de décorréler au maximum les signatures intermédiaires, mais de plus, la génération des références est facilitée car la signature intermédiaire après un départ en sous-programme ne dépend pas de la signature du sous-programme.

L'inconvénient majeur de cette technique est que le moniteur doit disposer d'une pile de signature de taille importante, ce qui grève le coût matériel dans le cas d'une implantation séparée. Dans le cas d'une implantation interne au processeur, il parait normal d'utiliser la pile du processeur ce qui se ressent au niveau du coût en performance.

1.4.4.2.4. Algorithmes de génération des références

La génération des références est le point clef des méthodes ESM avec ajustements car de l'efficacité de l'algorithme employé dépend le coût mémoire, le coût en performance et l'efficacité du dispositif de test. Le problème se décompose en fait en deux sous-problèmes. Le premier consiste à placer les singularités dans le programme source (assembleur), et le second consiste à calculer la signature intermédiaire en tout point du programme, à partir du code exécutable, afin d'en déduire les valeurs des singularités. Les deux types de singularités (ajustements et signatures de référence) nécessitent chacun un algorithme de placement particulier.

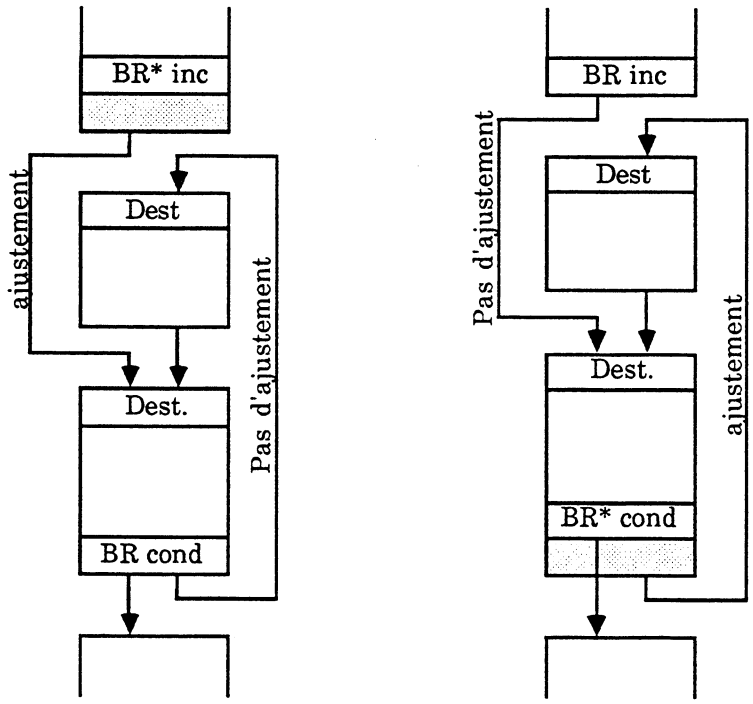
Pour le placement des signatures de référence, la latence de détection et le coût mémoire sont à considérer pour établir un compromis. Il faut remarquer que ce sujet n'a quasiment jamais été abordé dans la littérature. En particulier, il n'existe aucun algorithme permettant un placement des signatures de référence à latence garantie et coût mémoire minimum. Le coût en performance par contre est a priori toujours inversement proportionnel à la latence.

Le problème du placement des ajustements a été nettement plus étudié, en particulier ces derniers temps [Wart 90], [Wilk 91], [Schu 91], essentiellement pour diminuer le coût en performance lié à ces ajustements.

Indépendamment de l'aspect optimisation du coût en performance, le principal problème du placement des ajustements vient du fait qu'il faut, après placement des ajustements, être capable de calculer les signatures intermédiaires

en tout point du programme. C'est-à-dire qu'il faut trouver un moyen de relier toutes les instructions du programme en évitant les ajustements pour pouvoir propager le calcul de la signature intermédiaire. Ceci doit pouvoir être fait sans avoir recours au parcours systématique de tous les chemins du programme car sinon la complexité de l'algorithme peut devenir exponentielle [Wart 90]. L'existence de boucles dans un programme pose également un problème. En effet, une boucle doit a priori comporter au moins un ajustement pour assurer l'invariance de la signature et un placement systématique simpliste conduit parfois à des situations dans lesquelles le calcul des signatures intermédiaires est rendu très délicat.

Dans le cas d'ajustements sur arcs, un placement simpliste consiste à placer les ajustements sur les seuls arcs de séquencement conduisant à un point de jonction [Wilk 87]. Ceci conduit dans certains cas (Figure 1.13.a) à un placement incorrect des ajustements, car il peut exister des boucles sans ajustement et il peut être impossible d'arriver à certaines instructions par un chemin ne comportant pas d'ajustement.



(a) placement incorrect (b) placement correct

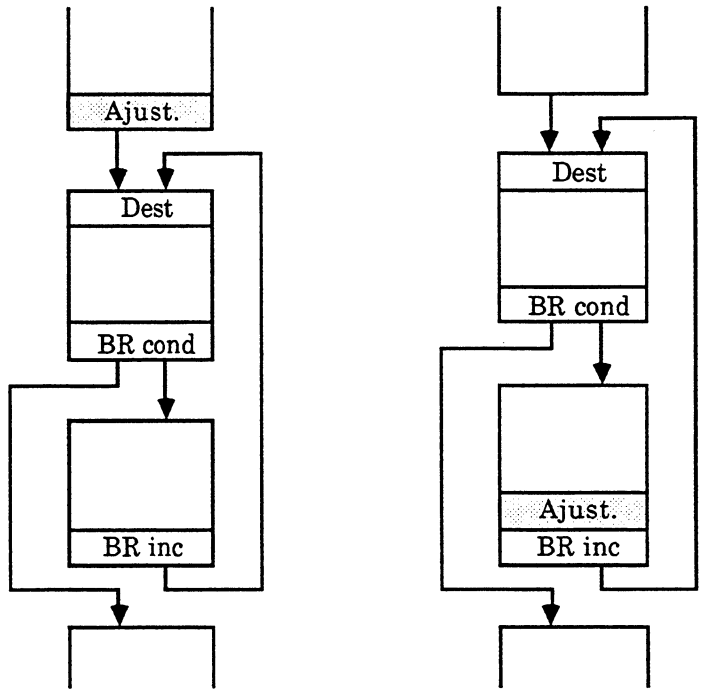
Figure 1.13 : problème du placement des ajustements sur arcs.

Dans le cas d'ajustements sur noeuds, un placement simpliste consiste à placer les ajustements dans les noeuds se terminant sur un point de jonction. Ceci

conduit également et pour les mêmes raisons que précédemment à un placement incorrect des ajustements (Figure 1.14.a).

Avec une fonction de compaction par code cyclique (MISR par exemple), il est théoriquement possible d'assurer l'invariance d'une boucle sans ajustements en imposant correctement une des signatures intermédiaires [Wilk 91]. Cependant, la génération des références est rendue très difficile car les boucles "auto-invariantes" doivent être identifiées, et leurs signatures intermédiaires doivent être calculées indépendamment et avant celles du reste du programme.

Une autre solution consiste à insérer systématiquement un ajustement par boucle mais il faut alors être capable d'identifier toutes les boucles d'un programme. L'algorithme proposé dans [Namj 82a] procède ainsi.



(a) placement incorrect (b) placement correct

Figure 1.14 : problème du placement des ajustements sur noeuds.

D'une manière générale, il faut éviter que certains blocs du programme soient "isolés" du reste par le fait que tous les chemins y conduisant aient un ajustement. Si le nombre d'ajustements est minimal, alors il existera forcément des boucles auto-invariantes à résoudre. Si le nombre d'ajustements n'est pas minimal, alors un certain nombre de signatures intermédiaires peuvent être choisies de manière arbitraire mais il faut identifier les instructions sur lesquelles cela doit être fait.

Dans le cas d'ajustements sur arcs, il existe une solution de facilité qui consiste à affecter un ajustement par instruction de séquencement, ce qui ne permet pas bien entendu d'atteindre le nombre minimal théorique d'ajustements. Avec ce mode de placement, toutes les destinations qui ne sont pas des points de jonction ont une signature intermédiaire arbitraire. Cette solution présente l'avantage de faciliter énormément le calcul des signatures intermédiaires car il suffit pour cela de parcourir linéairement l'ensemble du code.

Pour une méthode avec ajustements sur noeuds, une solution de facilité existe également pour le placement [Jay 86] mais à la différence des ajustements sur arcs, ceci ne facilite pas le calcul des signatures intermédiaires, au contraire, car il faut identifier les emplacements des signatures arbitraires.

Algorithme à coût en performance minimal [Wilk 91]

Pour diminuer le coût en performance lié aux ajustements, ceux-ci doivent être placés dans les parties de programme exécutées le moins fréquemment. Pour cela, chaque arc du graphe représentant le programme se voit affecter une valeur proportionnelle au coût d'un ajustement à cet emplacement, c'est-à-dire la fréquence d'exécution multipliée par le coût local de l'ajustement. Ce coût local dépend de la méthode utilisée (arcs ou noeuds) et de l'emplacement d'un ajustement.

Pour le placement des ajustements, un arbre reliant tous les sommets du graphe du programme est construit en prenant en priorité les arêtes dont le coût d'un ajustement est élevé (algorithme "glouton" de génération d'un arbre de poids maximum [Gond 85]). Les arêtes présentes dans le graphe et qui ne figurent pas dans l'arbre reçoivent un ajustement. En effet, si l'on ajoute une de ces arêtes à l'arbre ainsi créé, on génère un cycle, donc un point de jonction, ce qui nécessite un ajustement.

Pour la génération des signatures intermédiaires, il suffit de parcourir récursivement toutes les arêtes de l'arbre en propageant le calcul de la signature sur les sommets du graphe et en mémorisant la signature intermédiaire à chaque sommet.

Le calcul des valeurs d'ajustement consiste alors simplement à prendre toutes les arêtes présentes dans le graphe mais pas dans l'arbre et à leur affecter une valeur d'ajustement calculée à partir des signatures intermédiaires des deux sommets reliés par cette arête.

Remarques :

- Cet algorithme permet de placer le nombre minimal d'ajustements. En effet, pour un programme représenté par un graphe de n sommets, comportant N branchements conditionnels et pas de sous-programme ni de branchement à destinations multiples, le nombre d'arêtes dans ce graphe est égal à N+n-1. Comme un arbre reliant n sommets possède n-1 arêtes, le nombre d'ajustements à insérer est égal à N ce qui est le minimum absolu.

- Le parcours de l'arbre pour l'affectation des signatures intermédiaires à chaque sommet peut conduire à parcourir une arête correspondant à un bloc linéaire dans le sens opposé au sens d'exécution de ces instructions. Suivant la fonction de compaction utilisée, ce calcul est plus ou moins facile. Il est trivial dans le cas d'une compaction par OU Exclusif ou par addition, il reste faisable pour une compaction par MISR, mais la simplicité du calcul des signatures intermédiaires n'est pas le point fort de cet algorithme.

- Les boucles du programme comporteront obligatoirement un ajustement puisqu'une boucle est un cycle dans le graphe et que chaque cycle possède un ajustement. Cet algorithme fonctionne donc quelle que soit la fonction de compaction utilisée et sans qu'il soit nécessaire d'identifier toutes les boucles du programme. Dans [Wilk 91] il est également proposé un algorithme de placement optimal pour le cas particulier des fonctions de compaction à code cyclique qui autorisent la présence de boucles auto-invariantes.

- Les sous-programmes dont l'invariance est assurée sans mise en pile de la signature (cf 1.4.4.2.3) sont parfaitement pris en compte par cet algorithme, y compris dans le cas de sous-programmes récursifs.

Utilisations particulières de l'algorithme [Wilk 91]

L'algorithme précédent peut être utilisé pour générer le nombre minimal d'ajustements pour des schémas ne disposant que d'un seul mode d'ajustement (uniquement sur arcs ou bien uniquement sur noeuds). Pour cela, les arêtes sur lesquelles il n'est pas possible de faire un ajustement auront un coefficient non nul et celles sur lesquelles un ajustement est possible auront un coefficient nul.

Pour un placement du nombre minimal d'ajustements sur arcs, les arêtes ayant un coefficient nul sont celles correspondant à des arcs de séquencement, et les arêtes ayant un coefficient non nul sont celles correspondant à :

- des blocs linéaires,
- des retours de sous-programmes,
- des branchements à destinations multiples.

Pour un placement du nombre minimal d'ajustements sur noeuds, les arêtes ayant un coefficient nul sont celles correspondant à des arcs linéaires et les arêtes ayant un coefficient non nul sont celles correspondant à des arcs de séquencement.

1.4.4.2.5. Techniques de réduction de la latence de détection

Comme cela a déjà été mentionné, l'invariance de la signature par ajustements doit être complétée par l'ajout de singularités pour le test de la signature, ce qui augmente soit la latence de détection, soit le coût mémoire du dispositif de test. Certains auteurs ont donc proposé des mécanismes permettant de réduire la latence de détection sans augmentation du coût mémoire, c'est-à-dire sans introduction, dans le programme vérifié, d'informations dédiées aux tests. Ces techniques s'appliquent à n'importe quel schéma d'analyse de signature existant (et pas seulement ESM avec ajustements). Cependant, elles sont plus particulièrement utiles pour les méthodes avec ajustements et c'est pourquoi elles sont présentées avec ces méthodes. Le fait qu'elles aient été présentées toutes les deux avec un schéma d'ajustements sur arcs (cf 1.4.4.2.1) n'est pas tout à fait un hasard : l'introduction de références sur les arcs est un problème et donc la latence de ces méthodes est généralement élevée, ce qui conduit soit à rajouter des références sur certains noeuds, soit à utiliser une technique de réduction de la latence de détection comme celles décrites ici.

a/ Utilisation de signatures bidimensionnelles [Wilk 88]

L'emploi de "signatures bidimensionnelles" a été appliqué à un dérivé de la méthode [Wilk 87] et le tout a été présenté sous l'appellation CSM (Continuous Signature Monitoring) dans [Wilk 88] et [Wilk 90]. Le principe consiste à combiner une "signature verticale" avec une "signature horizontale". La signature verticale est la signature obtenue par compaction des instructions exécutées. La signature horizontale est la combinaison (par un OU Exclusif) d'une fonction de compaction (parité par exemple dans CSM) appliquée sur l'instruction seule avec un (des) bit(s) de la signature intermédiaire de l'instruction précédente. La signature horizontale ainsi obtenue est comparée à chaque lecture d'instruction avec une référence stockée dans un champ de bits rajouté à cet effet (Figure 1.15). Dans CSM, la mémoire de parité est utilisée pour le stockage de la référence horizontale.

Les erreurs détectables par la fonction de compaction horizontale sont détectées avec une latence nulle. Pour les autres erreurs, la latence moyenne de détection dépend du nombre de bits utilisés pour le stockage des références.

Soit h, le nombre de bits de la référence à chaque instruction (donc de la signature horizontale), la latence moyenne de détection est égale à [Wilk 88] :

$$L = \left(\frac{2^{-h}}{(1-2^{-h})} \right)^2$$

Cette latence décroît exponentiellement avec le nombre de bits de référence à chaque instruction. Dans CSM, à titre d'exemple, elle est égale à 1 cycle mémoire pour une référence sur 1 seul bit.

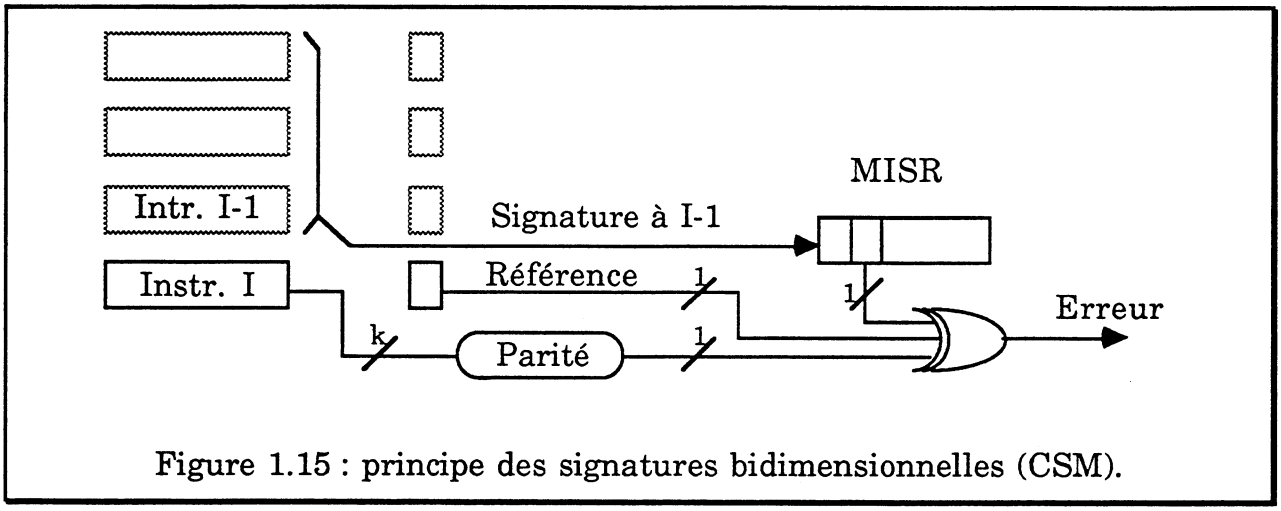


Figure 1.15 : principe des signatures bidimensionnelles (CSM).

Il faut remarquer qu'il est possible de combiner un bit d'étiquette identificateur de singularité avec une référence horizontale (hachage). Ainsi dans CSM, un identificateur haché permet de déterminer pour chaque branchement s'il doit effectuer ou non un ajustement (ajustements sur arcs). Ceci est une méthode astucieuse pour diminuer le nombre des ajustements sans augmenter le coût matériel de CSM, mais alors sur une instruction de branchement, le signal d'erreur (Figure 1.15) sert en fait à déterminer la présence d'un ajustement, ce qui dans certains cas peut introduire une augmentation de la latence de détection voire un masquage d'erreur.

b/ Utilisation de codes M parmi N [Upad 91]

Un code M parmi N (noté M/N) se définit comme étant l'ensemble des vecteurs de N bits ayant exactement M bits identiques. Cette propriété peut être testée par un dispositif matériel adéquat et il suffit donc d'un repérage (par mémoire étiquette par exemple) des instructions dont la signature intermédiaire est un code M/N pour réaliser le test de la signature sans l'introduction d'informations dans le programme.

Pour réduire les possibilités de masquage sur erreur de séquencement qui pourraient être introduites par cette méthode de vérification seule, il est proposé

dans [Upad 91] de faire un test à chaque point de jonction, donc de forcer un code M/N pour la signature intermédiaire de tous les points de jonction. Ceci n'est possible qu'en augmentant le nombre d'ajustements par rapport au nombre théorique minimal, ce qui est le cas dans [Upad 91] car les ajustements sont faits sur tous les arcs, laissant ainsi un certain nombre de signatures arbitraires.

La latence de détection ainsi obtenue dépend du code M/N choisi et elle peut être réduite en vérifiant que les signatures intermédiaires des instructions non étiquetées ne sont pas des codes M/N. Une latence moyenne de 3 à 4 instructions est avancée dans [Upad 91] ce qui est assez faible.

1.4.4.3. Conclusion sur les méthodes ESM

L'avantage des méthodes ESM vient du faible coût mémoire pour le stockage des informations de test, de la faible complexité matérielle du dispositif d'analyse de signature et de la possibilité d'intégrer très avantageusement le moniteur directement dans le processeur.

Leur inconvénient est l'obligation de modifier systématiquement le programme vérifié, avec tous les risques d'introduction d'erreurs que cela comporte ainsi qu'un ralentissement de l'exécution du programme vérifié. Par ailleurs, la modification du code doit être faite en deux phases. La première consiste à insérer des singularités dans le source assembleur et la seconde consiste à calculer la valeur de ces singularités à partir du code exécutable. Il est donc nécessaire de disposer d'un assembleur spécial mais également d'un éditeur de liens modifié et la génération des références d'un programme dont on ne possède pas le source nécessite son désassemblage.

L'efficacité des méthodes ESM dépend de la manière d'assurer l'invariance de la signature. Les méthodes avec ajustements assurent une meilleure détection des erreurs de séquencement à travers la signature car celle-ci est représentative des chemins suivis par le processeur.

Pour toutes les méthodes ESM, il faut s'assurer que les cas où le processeur ne rencontre jamais d'instruction de test (sortie du code par exemple) sont détectés. Ceci peut être fait grâce à certaines caractéristiques de la méthode elle-même (PSAb, [Saxe 89]) ou alors par un dispositif annexe. Ce dispositif peut être un classique chien de garde temporel et dans ce cas, il doit être rafraîchi par un test de signature positif. Le dispositif annexe peut aussi être une détection de violation des frontières du code, technique de test en ligne déjà très puissante à elle seule. Cette détection peut être assurée soit par comparaison des adresses accédées avec les frontières de la zone contenant le code, comme dans SIS, soit par l'utilisation d'une mémoire étiquette pour la détection des branchements erronés comme dans PSAb.

L'utilisation d'une technique de réduction de la latence de détection comme dans CSM apporte les mêmes bénéfices.

Le faible coût matériel du moniteur des méthodes ESM ne doit donc pas occulter le fait qu'il est nécessaire, dans de nombreux cas, de compléter l'analyse de signature par un dispositif supplémentaire.

1.4.5. Méthodes DSM

Les méthodes DSM ont été proposées pour remédier à deux défauts majeurs des méthodes ESM qui sont :

- une modification systématique du programme vérifié,
- une baisse de performance dans l'exécution du programme vérifié.

Par ailleurs, certains auteurs ont proposé comme avantages des méthodes DSM, l'utilisation possible d'un seul processeur de vérification pour plusieurs processeurs, ainsi qu'une génération des références par apprentissage. Ceci ne peut être réalisé qu'avec des méthodes DSM mais reste optionnel et il ne faudrait pas assimiler certaines méthodes DSM à leur implantation multi-processeurs ou à l'apprentissage pour la génération des références.

La génération des références est beaucoup plus facile pour les méthodes DSM que pour les méthodes ESM. En particulier, une seule phase est nécessaire et elle peut être effectuée directement sur le code exécutable. Ceci permet de vérifier des programmes sans avoir à les re-compiler, ainsi que du code dont on ne possède pas le source. Ceci est particulièrement appréciable dans le cas des bibliothèques d'un langage par exemple.

1.4.5.1. Méthode Cerberus-16 [Namj 83]

Cette méthode a été proposée comme une solution pour supprimer le coût en performance des méthodes ESM, en particulier de PSA, seule technique de vérification d'un flot de contrôle par analyse de signature connue à cette époque. Cerberus-16 constitue un cas très particulier dans l'ensemble des méthodes d'analyse de signature car la signature n'est jamais ni ajustée ni réinitialisée et donc l'invariance de la signature aux points de test n'existe pas. La signature du programme est générée uniquement à partir de la suite des instructions lues par le processeur. Le rôle du watchdog Cerb.16 consiste simplement à recréer la signature du processeur à partir d'une description du graphe du programme vérifié. Cette description comporte une instruction watchdog pour chaque bloc linéaire et la structure du graphe du watchdog est exactement la même que celle du programme vérifié. Par abus de langage on assimilera donc le terme de "noeud" à une instruction watchdog et donc au bloc linéaire correspondant.

Le format des instructions (noeuds) Cerb.16 est le suivant :

champ 1 : type du noeud : type de la dernière instruction du bloc linéaire correspondant,

champ 2 : taille du noeud : nombre de mots (ou d'instructions) dans le bloc linéaire correspondant,

champ 3 : signature du bloc linéaire,

champ 4 : adresse du noeud successeur dans le programme watchdog. Ce champ n'est présent que si le type du noeud correspond à un branchement.

Le repérage des singularités est exclusivement de type relatif : pour déterminer l'occurrence d'un noeud, le watchdog compte le nombre d'instructions lues depuis la première instruction du noeud et lorsque ce nombre correspond au champ 2, le watchdog considère que le processeur lit la dernière instruction du noeud. Le début d'un noeud est l'instruction exécutée immédiatement après l'instruction de fin d'un noeud.

Le fonctionnement du watchdog consiste à émuler la génération de la signature du programme vérifié. Pour cela, il doit suivre le cheminement du processeur grâce aux informations sur la structure du graphe et recréer la signature à partir des informations de signature relatives à chaque noeud. La signature du programme et la signature recréée par le watchdog sont donc identiques à chaque noeud et un test peut donc avoir lieu à ces endroits. Il faut remarquer que la signature du programme n'est pas comparée à une signature de référence mais à la signature recréée par le watchdog à partir d'une liste de références.

Remarques :

Ce type de vérification s'apparente tout à fait au principe de "l'observateur" décrit dans [Ayac 79], l'abstraction vérifiée se rapportant au flot de contrôle du processeur dans le cas de Cerberus 16.

Le watchdog fonctionne exactement de la même manière que le processeur vérifié en ce qui concerne les instructions de séquencement. En particulier, lors des départs et retours de sous-programme, le watchdog doit sauver et restituer son pointeur d'instructions dans sa propre pile.

Contrairement à la plupart des schémas d'analyse de signature, la fonction de compaction n'est pas indifférente. En effet, il faut que la signature de référence affectée à un noeud produise le même effet sur la signature que la compaction de la suite des instructions du bloc, et ce quelle que soit la signature d'entrée du bloc

linéaire. Ceci vient du fait que l'invariance de signature n'est pas respectée. La relation que doit vérifier la fonction de compaction est la suivante :

Compaction(Se,Sn) = Compaction(Se,I0,....IN), ∀ Se {A}

avec Se = signature d'entrée du bloc linéaire,
Sn = Compaction(I0,.... IN) = signature affectée au noeud,
I0 à IN = codes des instructions du bloc linéaire.

On doit donc exclure d'office la division polynomiale. Une compaction par OU Exclusif ou par addition est proposée dans [Namj 83] ainsi qu'une troisième fonction de compaction (décalage et OU Exclusif). Cette dernière fonction ne respecte pas la relation {A}.

La détection des ruptures de séquence inopinées est possible avec Cerb.16 car le watchdog connaît l'emplacement des instructions de séquencement. Les branchements erronés par contre ne sont pas détectables instantanément à cause du repérage relatif des singularités. La détection interviendra par le fait que le programme exécuté par le processeur ne correspondra pas au graphe de référence.

Le coût mémoire de la méthode est très élevé car il faut stocker trois ou quatre informations (2 ou 3 mots mémoire) par instruction watchdog c'est-à-dire par bloc linéaire, là ou PSAb se contente d'un seul mot mémoire par exemple.

1.4.5.2. Méthode ASIS [Eife 84]

ASIS, pour "Asynchronous Signatured Instruction Stream", a été proposée comme un dérivé de SIS (cf 1.4.4.1.2) permettant une vérification multi-processeurs L'implantation de cette méthode à été présentée dans [Toma 85] sous l'appellation RVMP (RoVing Monitoring Processor). La méthode reste bien évidemment applicable pour un seul processeur. Le processeur de vérification dispose, comme dans Cerb.16, d'une image du graphe du programme vérifié avec une information de signature à chaque noeud du graphe image. La décomposition du programme en noeud est par contre très différente et assez originale. Dans cette représentation, un noeud est associé non pas à un bloc linéaire mais à un chemin linéaire (Définition 1.6, cf 1.1.1) qui peut comporter des instructions de branchement inconditionnel ainsi que des points de jonction.

Un chemin linéaire se termine toujours par une instruction ayant plusieurs destinations (branchement conditionnel, retour de sous-programme, branchement à destinations multiples), et il commence toujours par la première instruction exécutée après une fin de chemin linéaire (destination d'un branchement conditionnel ou à destinations multiples, instruction située après un branchement conditionnel ou un départ en sous-programme). Si la première instruction d'un

chemin linéaire est exécutée alors la dernière sera également exécutée. Une signature unique peut donc être associée à un chemin linéaire à condition que la signature d'entrée soit unique également. Dans le graphe représentant le programme vérifié, la signature d'un chemin linéaire est liée à un noeud correspondant à ce chemin linéaire. Les arcs sortant d'un noeud vont vers les noeuds qui représentent les chemins linéaires légaux après ce noeud et chaque noeud possède donc autant de successeurs que l'instruction terminant le chemin a de destinations, par exemple deux dans le cas d'un branchement conditionnel. A partir de cette représentation, le watchdog suit le cheminement du processeur grâce à la signature : pour déterminer le chemin emprunté par le processeur, le watchdog attend la fin d'un chemin linéaire et regarde si la signature de ce chemin figure dans la liste des successeurs possibles du noeud courant. Il prend alors comme noeud courant le noeud dont la signature de référence correspond à la signature du chemin qui vient d'être exécuté. Une erreur de séquencement ou de signature sera détectée si la signature obtenue à la fin d'un chemin ne correspond à aucune destination possible dans le graphe de référence.

Il faut remarquer qu'un bloc linéaire commençant par un point de jonction appartient forcément à plusieurs chemins linéaires. La signature étant calculée sur l'ensemble d'un chemin linéaire sans ajustements ni réinitialisation aux points de jonction, il existe autant de signatures intermédiaires à une instruction que de chemins linéaires passant par cette instruction. La méthode ASIS ne respecte donc pas l'invariance des signatures intermédiaires en tout point du programme. La signature d'un chemin linéaire par contre doit être invariante et comme un noeud (chemin linéaire) peut être atteint depuis différents chemins linéaires qui ont des valeurs de signature différentes, il faut réinitialiser la signature en début de chemin linéaire avec une valeur fixe identique pour tous les chemins linéaires.

En ce qui concerne les possibilités de masquage, aucune étude n'a été faite mais on peut facilement déduire du fonctionnement du watchdog que cette méthode est a priori moins efficace que les méthodes testant explicitement une signature invariante. D'une part la signature n'est jamais comparée à une référence unique mais à une liste de signatures possibles et d'autre part, la réinitialisation du dispositif de compaction en début de chemin entraîne une corrélation des signatures intermédiaires d'autant plus forte que le nombre de signatures intermédiaires est supérieur au nombre d'instructions à cause de la non invariance. En particulier, la probabilité que deux chemins linéaires distincts aient la même signature n'est pas du tout négligeable. D'autre part, les erreurs propres du watchdog peuvent également être masquées car une même instruction de fin de chemin correspond à plusieurs noeuds dans le graphe du watchdog.

Le coût mémoire est très élevé d'une part en raison de la quantité importante d'informations à stocker pour chaque instruction du watchdog et d'autre part à cause du nombre d'instructions watchdog. Ce coût mémoire est assez complexe à calculer. Il est égal à deux fois le nombre d'états ayant deux successeurs (branchements conditionnels) dans une structure de programme ne comportant que des noeuds ayant au plus deux successeurs.

1.4.5.3. Méthode OSLC [Made 91a]

Cette méthode est identique à PSAb (cf 1.4.4.1.1) en ce qui concerne la génération de la signature du processeur vérifié : la signature est initialisée à une valeur fixe en début de bloc et elle est testée en fin de bloc. Comme pour PSAb, il est donc nécessaire de stocker une signature de référence par bloc linéaire mais, contrairement à PSAb qui est de type ESM, ceci est fait pour OSLC dans une mémoire séparée du programme. Le repérage des extrémités de blocs est aussi légèrement différent mais tout à fait équivalent. La différence a pour but manifeste de réduire la taille du champ de bits d'étiquette : un bit d'étiquette unique est utilisé pour signaler le début des blocs commençant par un point de jonction (blocs de type A ou C, cf figure 1.3). Le décodage des instructions de séquencement signale la fin des blocs terminés par une telle instruction (blocs de type C ou D, cf figure 1.3). Les autres extrémités sont repérées relativement car elles sont toujours situées immédiatement à coté (plus ou moins une instruction) d'une extrémité repérée. Ce mode de repérage des singularités permet comme dans PSAb de détecter les ruptures de séquence inopinées ainsi que les branchements hors de la zone mémoire contenant du code. Pour le test de la signature, une méthode originale est employée : le programme est décomposé en segments associés chacun à un nombre donné de signatures de références. Ces segments sont repérés par leur adresse de début et leur adresse de fin. Une signature sera considérée juste si elle correspond à une référence associée au segment auquel appartient l'instruction de fin de bloc sur laquelle a lieu le test. Le processeur de vérification ne dispose donc pas d'une image du graphe du programme vérifié ce qui permet de réduire le coût mémoire des références d'autant plus fortement que le nombre de segments est faible, c'est-à-dire que la taille des segments est importante. Ce mode de test par contre, introduit une possibilité de masquage d'autant plus importante que le nombre de signatures par segment est grand donc que la taille des segments est importante. Un compromis est donc à faire entre masquage et coût mémoire. Par ailleurs, comme dans toutes les méthodes qui réinitialisent la signature en début de bloc (cf 1.4.4.1.4), un branchement erroné dont la destination est un début de bloc ne sera pas détecté.

Pour la génération des références, il est proposé dans [Made 91a] une méthode d'apprentissage. Ceci est très séduisant sur le principe et son auteur considère qu'il

s'agit là du point fort de sa méthode. Mais pour que l'apprentissage génère toutes les références, il faut que tous les arcs du graphe représentant le programme soient exécutés au moins une fois. En particulier, tous les branchements conditionnels doivent être exécutés au moins une fois avec leur condition vraie et au moins une fois avec leur condition fausse. Pour un programme comportant N branchements conditionnels, il faudra, pour satisfaire cette condition, exécuter le programme un nombre de fois compris entre N+1 et 2^N. Cet apprentissage constitue donc en fait le point le plus critiquable de la méthode.

Il reste bien évidemment possible d'utiliser OSLC avec une génération de références classique, par un programme séparé, et dans ce cas la génération est particulièrement simple car l'invariance de la signature est assurée par réinitialisation du dispositif de compaction.

Cette méthode a été proposée avec une implantation pour la vérification multi-processeurs. Dans ce cas, les signatures des blocs sont envoyées au processeur de vérification avec l'adresse de fin de bloc pour permettre le test qui est fait par corrélation entre signature et adresse.

1.4.5.4. Vérification multi-processeurs

Le processeur watchdog utilisé dans une méthode DSM n'utilise en général qu'une très petite fraction de son potentiel. Son rôle consiste en effet seulement à accéder aux références puis à comparer ces références avec les informations du dispositif de génération de signature à chaque singularité nécessitant un test. D'où l'idée naturelle de partager ce watchdog pour plusieurs processeurs disposant chacun d'un dispositif de génération de signature et d'un repérage des singularités du programme. Pour éviter que les processeurs vérifiés ne soient ralentis, une file de signature est placée entre le dispositif de génération et le watchdog de vérification (Figure 1.16). Ce type de réalisation a été proposé dans [Toma 85] pour la méthode ASIS (processeur RVMP) et dans [Made 91a], [Made 91b] pour la méthode OSLC.

L'idée de la vérification simultanée de plusieurs processeurs est séduisante sur le principe mais en fait il est probable que le coût réel d'une telle implantation soit supérieur à celui où chaque processeur dispose de son propre watchdog de vérification. En effet, un watchdog permettant la vérification de plusieurs processeurs en même temps est beaucoup plus complexe qu'un watchdog ne vérifiant qu'un seul processeur. Par exemple, la réalisation du RVMP [Toma 84] nécessite 6000 portes équivalentes uniquement pour le processeur de vérification.

Il ne faut pas oublier que chaque processeur vérifié doit disposer d'un générateur de signature comprenant un dispositif de compaction et un repérage des singularités, que le watchdog soit commun à plusieurs processeurs ou non.

Dans le cas d'une vérification multi-processeurs, il faut de plus rajouter le coût associé à la file de signature (FIFOs double accès, interfaces du watchdog et du générateur avec la file).

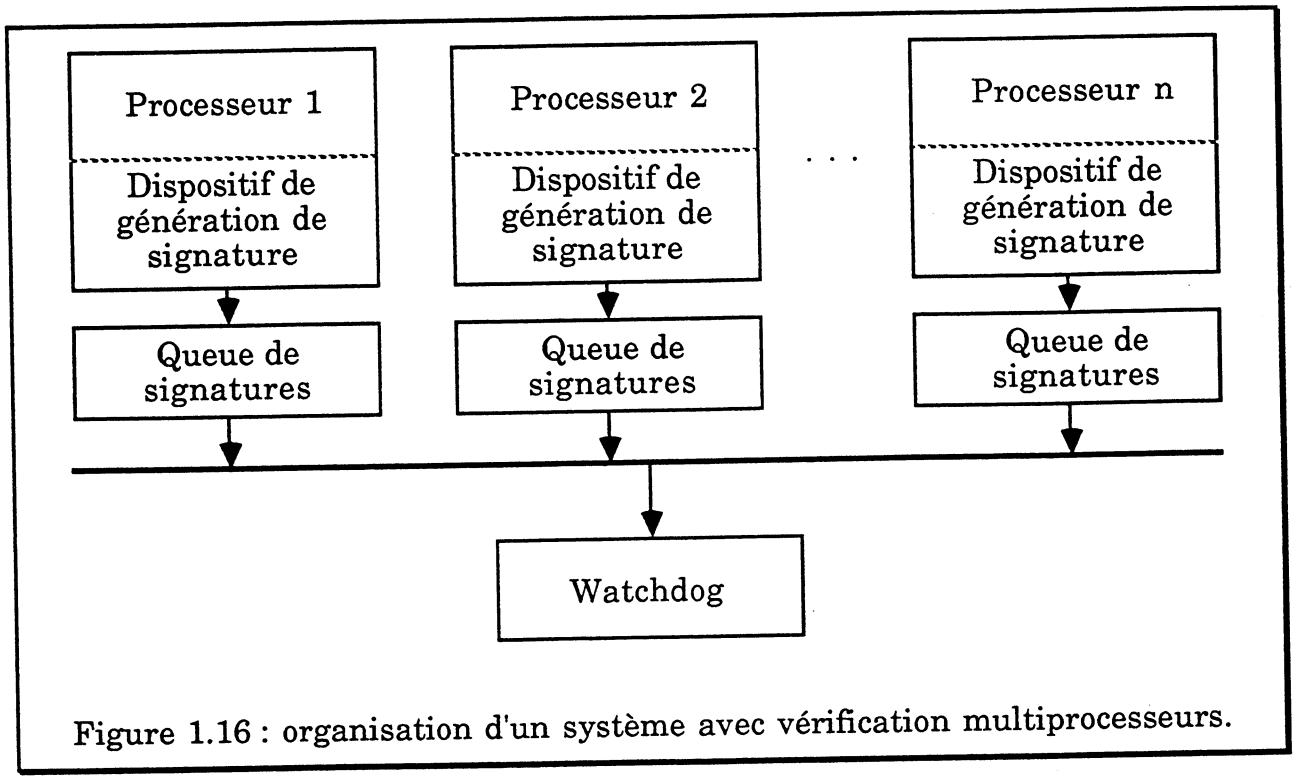


Figure 1.16 : organisation d'un système avec vérification multiprocesseurs.

L'implantation multiprocesseurs peut donc difficilement être considérée comme un argument en faveur des méthodes DSM, car un système multiprocesseurs dans lequel chaque processeur est vérifié indépendamment par un moniteur ESM sera toujours beaucoup plus économique que le même système vérifié par un seul watchdog DSM.

1.4.5.5. Conclusion sur les méthodes DSM

Les méthodes DSM ont les avantages suivants par rapport aux méthodes ESM :
- pas de modification systématique du programme application,
- coût en performance théoriquement nul.

De plus, les méthodes DSM présentées jusqu'à maintenant ne nécessitent pas de dispositif complémentaire pour détecter les cas où la signature n'est jamais testée, contrairement à la plupart des méthodes ESM. Ceci vient du fait que ces méthodes n'utilisent pas d'ajustement et que le repérage des singularités est obligatoirement fait d'une manière externe au programme vérifié.

Notons qu'aucune méthode DSM n'a jusqu'ici été proposée avec des ajustements alors que dans le cas des méthodes ESM, ceci permet de réduire sensiblement le coût mémoire des informations nécessaires à la vérification.

Par ailleurs, les méthodes DSM permettent de faire de manière optionnelle un certain nombre d'actions (même si leur intérêt n'est pas toujours évident) peu envisageables avec des méthodes ESM :

- génération des références par apprentissage,
- vérification de plusieurs processeurs par un seul watchdog,
- pas d'invariance de la signature aux points de test.

L'inconvénient majeur des méthodes DSM vient de leur coût mémoire supérieur aux méthodes ESM. En effet, en plus des informations relatives à la signature, le programme du watchdog doit contenir des informations sur la structure du programme vérifié. Le coût matériel est aussi en général plus élevé car le watchdog doit accéder à une mémoire d'une manière autonome pour ne pas dégrader les performances du système. Par ailleurs, l'intégration d'un watchdog DSM au processeur vérifié ne présente quasiment aucun intérêt contrairement aux moniteurs des méthodes ESM pour lesquels l'intégration constitue la solution idéale.

On pourra également remarquer que les méthodes DSM proposées jusqu'à présent sont toutes très différentes et assez originales par rapport aux méthodes ESM, mais qu'elles ont cependant toutes un point commun très critiquable : dans ces méthodes, la signature du processeur n'est jamais comparée avec une référence unique, précalculée et mémorisée comme telle. La signature est comparée soit à une liste de références possibles (ASIS et OSLC), soit à une signature recréée par le watchdog (Cerb.16).

1.4.6. Conclusion sur le test au niveau programme

Un très grand nombre de méthodes de vérification au niveau programme ont déjà été proposées. Elles se divisent essentiellement en deux catégories, ESM et DSM suivant le mode de stockage des informations pour le test. Les méthodes ESM utilisent toutes le principe de l'invariance de la signature en tout point du programme, ceci afin de faciliter la vérification. On peut ainsi distinguer deux catégories de méthodes ESM suivant que l'invariance de la signature est obtenue par ajustements ou par réinitialisation du dispositif de compaction aux points de jonction. Une telle classification n'est pas possible sur les méthodes DSM qui sont toutes très différentes et qui ne respectent pas, pour certaines, l'invariance de la signature en tout point du programme. Une synthèse des principales caractéristiques des méthodes présentées dans ce chapitre est donnée dans le tableau 1.1 .

Les méthodes ESM ont été les plus étudiées car leur faible coût matériel est séduisant, mais elles entraînent une baisse des performances du système. A

l'inverse, les méthodes DSM sont généralement plus coûteuses d'un point de vue matériel mais elles n'introduisent pas de baisse de performance du système et ne nécessitent pas de modification des programmes vérifiés.

Tableau 1.1 : synthèse des méthodes de vérification d'un flot de contrôle au niveau programme par signature dérivée.

Méthode	Référence	Type	Invariance	Singularités	Repérage des singularités	Vérification imposée	Fonction de compaction
PSAb	[Namj 82a]	ESM	réinit.	extr. blocs	2 bits étiq.	fin de blocs	XOR
PSAg	[Namj 82a]	ESM	réinit. + ajs noeuds	ajoutées	2 bits étiq.	non	XOR
Cerberus16	[Namj 83]	DSM	non	extr. blocs	relatif	fin de blocs	XOR*
SIS	[Schen 83]	ESM	réinit.	extr. blocs	décodage	non	div. pol.
ASIS	[Eife 84]	DSM	non	divergences	décodage	divergences	div. pol.
Hsurf	[Jay 86]	ESM	ajs noeuds	ajoutées	décodage	non	div. pol.
---	[Wilk 87]	ESM	ajs arcs	instr. séq.	décodage	retours sp	div. pol.
CSM	[Wilk 88]	ESM	ajs arcs	instr. séq.	décodage + 1 bit étiq.	continue	div. pol. + parité
---	[Saxe 89]	ESM	réinit.	extr. blocs	décodage	fin de blocs	addition*
---	[Upad 91]	ESM	ajs arcs	instr. séq.	décodage + 1 bit étiq.	code M/N	div. pol.
OSLC	[Made 91a]	DSM	réinit.	extr. blocs	décodage + 1 bit étiq	fin de blocs	div. pol.

Au niveau de l'implantation matérielle, toutes les méthodes ont en commun un dispositif de compaction, qui est le plus souvent un MISR, et un dispositif de repérage des singularités.

Le choix de la fonction de compaction ne dépend en général que de considérations liées à la probabilité de masquage, sauf dans certains cas très particuliers où les propriétés algébriques de la fonction de compaction ont une importance [Saxe 89] et Cerb. 16 [Namj 83] (*).

Le choix du mode de repérage des singularités est très important car il influence directement plusieurs facteurs du dispositif de test comme le coût matériel, le coût mémoire et le pouvoir de détection pour certains types d'erreurs. Tous les modes de repérage ne s'appliquent pas à tous les schémas d'analyse de signature ni à tous les types de processeurs vérifiés.

En ce qui concerne la génération des références, les méthodes avec ajustements nécessitent des algorithmes complexes pour le placement des ajustements et un compromis est souvent à faire entre le coût mémoire, le coût en

performance, la facilité de calcul des signatures intermédiaires et la possibilité d'un calcul modulaire. Par ailleurs, aucun algorithme de calcul des références n'est actuellement en mesure de faire un placement des points de test à latence garantie en respectant d'autres critères d'optimisation.

A l'inverse, pour les méthodes qui réinitialisent la signature aux points de jonctions, le placement des singularités est implicite, la latence de détection est fixe, et le calcul des signatures intermédiaires en tout point du programme est trivial, mais dans certains cas, la corrélation des signatures intermédiaires qui en résulte peut provoquer une diminution du pouvoir de détection.

Le calcul des informations pour le test dans le cas des méthodes ESM doit être fait en deux phases. Une première phase place les singularités dans le source (avant assemblage) et une deuxième phase calcule la valeur des singularités à partir du code exécutable (après assemblage/édition de liens).

A l'inverse, pour les méthodes DSM, une seule phase de calcul à partir du code exécutable suffit pour le calcul des informations pour le test. Il est donc beaucoup plus aisé de calculer des références sur du code dont on ne possède pas le source.

Le choix d'un schéma particulier de test au niveau programme dépend donc de considérations qui doivent porter sur l'ensemble d'un système et sur les possibilités de détection attendues pour le dispositif de test (latence en particulier). Au niveau du coût d'implantation d'une méthode dans un système donné, certaines solutions sont préférables globalement, même si, sur certains points précis, d'autres solutions peuvent présenter des caractéristiques plus avantageuses.

1.5. Test au niveau graphe d'états d'une machine câblée

Plusieurs méthodes ont été proposées pour vérifier le fonctionnement d'un contrôleur de circuit VLSI [Srid 82a], [Namj 82c], [Iyen 85], [Tung 86]. Toutefois, celles-ci sont quasi exclusivement consacrées aux séquenceurs microprogrammés construits autour d'une ROM. Dans ce cas, certaines techniques utilisées au niveau système peuvent être employées en remplaçant la notion de programme par celle de microprogramme [Namj 82c]. Peu d'études ont cependant été consacrées aux autres type d'implantations de contrôleurs.

Pour le test d'un contrôleur câblé avec codage compact[1], une méthode originale a été proposée dans [Leve 90a], [Leve 90b]. Cette méthode s'adresse essentiellement à des séquenceurs synchrones, ayant un graphe bien structuré et dans lesquels il

---

1 Codage des états sur un nombre minimum de bits.

n'existe aucun parallélisme, c'est-à-dire des contrôleurs dans lesquels des chemins peuvent être facilement identifiés et où un seul état est actif à un instant donné. Les études ultérieures [Holm 91], [Robi 92] ont également montré l'intérêt de l'approche dans le cas de graphes peu structurés.

1.5.1. Principe de la méthode

1.5.1.1. Introduction d'une propriété invariante dans le graphe [Leve 90a]

La stratégie de base consiste à tester une signature obtenue à partir du GFC du contrôleur. Pour cela, les codes des états atteints par le contrôleur sont successivement compactés par un MISR. Ce registre contient ainsi à chaque cycle d'horloge une signature représentant le chemin suivi à travers le GFC du contrôleur. Aux points de vérification (états critiques du graphe, par exemple), la signature courante est comparée à une référence ; toute différence permet alors de détecter un chemin illégal. Cette comparaison ne peut s'effectuer facilement que si la référence associée à un point de vérification donné est unique (indépendante du chemin suivi dans le graphe et de l'instant de vérification). Il est donc nécessaire de forcer dans le graphe une propriété invariante sur les signatures des séquences de codes d'états.

Propriété P1 (invariance forcée) : la signature d'une séquence de codes d'états obtenue par division polynomiale, avec un polynôme diviseur donné, en suivant un chemin légal d'un GFC, est invariante en sortie de chaque état de ce GFC.

Cette propriété est identique à l'invariance des signatures intermédiaires telle qu'elle est considérée au niveau programme (cf 1.4.2) mais en remplaçant la notion de signature à une instruction du programme par la notion de signature à un état du graphe. De même, la signature utilisée comme référence lors d'une comparaison est la signature invariante obtenue au niveau d'un point de vérification.

Obtenir la propriété P1 revient à satisfaire la condition nécessaire et suffisante suivante :

Condition nécessaire et suffisante : les signatures des séquences de codes d'états obtenues par division polynomiale, avec un polynôme diviseur donné, en suivant des chemins légaux du GFC, sont identiques après tous les états ayant un successeur commun.

Une conséquence intéressante de cette condition peut être dérivée dans le cas d'un état rebouclant sur lui-même :

Condition nécessaire : les signatures des séquences de codes d'états obtenues par division polynomiale, avec un polynôme diviseur donné, sont identiques avant et après un état rebouclant sur lui-même.

Pour satisfaire la propriété P1, la méthode des ajustements telle quelle est utilisée au niveau programme (cf 1.4.4.2) présente l'inconvénient de nécessiter la mémorisation des valeurs d'ajustement, ce qui se traduit par une augmentation de matériel et de ce fait, cette technique a été écartée au profit d'une méthode intervenant directement sur les données compactées. En d'autres termes, le choix correct des codes attribués aux états est effectué de telle sorte que l'invariance des signatures intermédiaires soit assurée sans faire intervenir d'ajustements explicites, donc sans nécessiter le stockage d'informations.

Le problème revient à trouver un polynôme diviseur, des codes d'états et un ensemble de signatures tels que ces contraintes soient satisfaites.

Définition 1.15 : on appelle "S-codage" d'un graphe de contrôle un codage des états rendant la signature des séquences de codes d'états invariante après chaque état.

Satisfaire la propriété P1 lors du codage des états impose de sévères contraintes sur ce codage. Ces contraintes peuvent, dans la plupart des cas, être satisfaites lorsque le graphe du séquenceur est bien structuré. Il existe cependant des cas où la structure du graphe ne permet pas le S-codage. Ceci nous amène à distinguer deux types de graphes.

Définition 1.16 : on appelle "SC-graphe" un graphe de contrôle dont la structure autorise un S-codage. On appelle "NSC-graphe" un graphe de contrôle n'étant pas un SC-graphe.

1.5.1.2. Principe de la vérification

Un ensemble d'états est choisi, sur lesquels la signature courante doit être vérifiée (points de vérification). Ce choix est effectué en fonction des contraintes sur la latence de détection d'erreur et du coût autorisé pour l'implantation du test en ligne. Le principe consiste à stocker les références associées aux points de vérification et à effectuer, lorsqu'un de ces points est atteint, la comparaison de la signature courante et de la référence correspondant au point atteint. En chaque point de vérification sont générés des signaux de contrôle utilisés pour la commande de la comparaison et la sélection de la bonne référence parmi celles stockées.

1.5.2. Implantation de contrôleurs S-codés

Trois problèmes doivent être successivement abordés pour implanter un contrôleur vérifié par analyse de signature suivant le principe de la section 1.5.1 :

- déterminer si un graphe de contrôle est un SC-graphe,
- transformer (le cas échéant) un NSC-graphe en un SC-graphe,
- obtenir un S-codage du SC-graphe.

L'analyse et la résolution de ces trois problèmes sont traitées dans [Leve 90a] selon l'approche générale résumée sur la figure 1.17 .

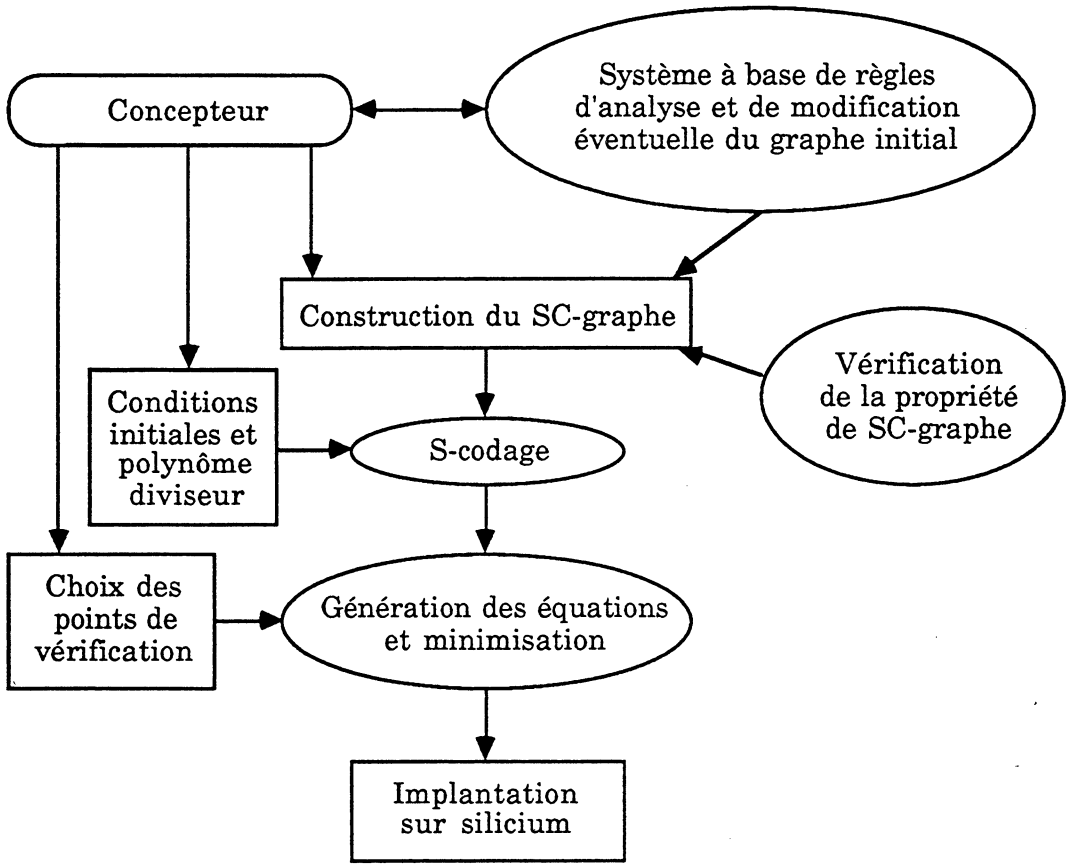


Figure 1.17 : méthode de conception d'un contrôleur S-codé [Leve 90a].

L'obtention d'un SC-graphe est la phase critique de la conception d'un contrôleur S-codé car il est rare qu'un graphe soit codable dans sa description originale si le concepteur n'a pas pris en compte dès le début les contraintes liées à l'analyse de signature. La transformation d'un NSC-graphe en un SC-graphe est une phase délicate qui peut nécessiter une intervention du concepteur pour atteindre un surcoût raisonnable pour le test en ligne. Une méthode de transformation entièrement automatique a aussi été proposée dans [Robi 92].

La meilleure solution consiste toutefois, lorsque ceci est possible, à prendre en compte les contraintes liées à la S-codabilité dès le début de la conception du graphe du contrôleur. En particulier, certaines situations dans un graphe conduisent à devoir imposer le même code à deux états différents ce qui est bien entendu impossible.

Définition 1.17 : Deux états sont dits "C-équivalents" dans un graphe de structure donnée si cette structure et le respect de la propriété P1 conduisent à imposer le même code pour les deux états quels que soient le polynôme diviseur premier primitif et les conditions initiales de codage choisis.

Le rôle du concepteur sera donc d'éviter systématiquement les situations conduisant à la C-équivalence de deux états, tout en respectant la fonctionnalité requise pour le contrôleur. Les principales structures conduisant à la C-équivalence de deux états sont présentées dans la figure 1.18 .

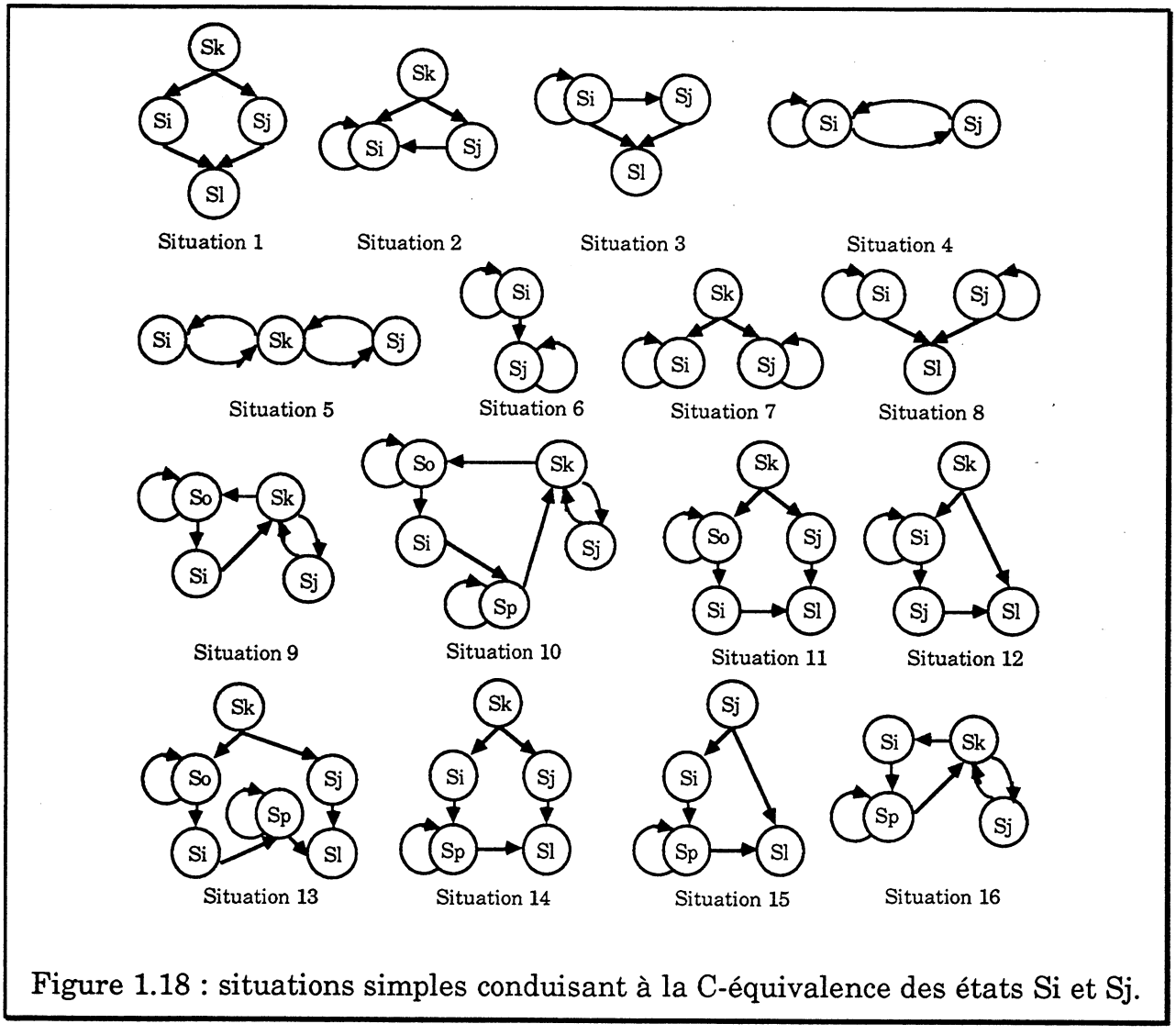


Figure 1.18 : situations simples conduisant à la C-équivalence des états Si et Sj.

La liste de ces situations n'étant pas exhaustive et leur identification dans un graphe pas toujours aisée, l'obtention d'un SC-graphe nécessitera tout de même, en général, une transformation de la description originale mais beaucoup plus aisée si les principales situations ont été évitées. L'exemple du contrôleur S-codé du circuit HSURF32 (cf 1.6.3) illustre parfaitement ceci.

1.6. HSURF32 : un exemple de vérification à deux niveaux

Le circuit HSURF32 est un microprocesseur qui, dans ses dernières versions, intègre des dispositifs de vérification du flot de contrôle à deux niveaux. Au niveau programme, un moniteur ESM (avec ajustement sur noeuds) est implanté au sein du microprocesseur (cf 1.6.2). Au niveau circuit, le contrôleur est réalisé suivant la méthode de S-codage décrite en section 1.5.2 (cf 1.6.3).

1.6.1. Introduction

Le projet HSURF (Haute SUreté de Fonctionnement) de conception de microprocesseurs à test en ligne intégré prend ses origines dans [Jay 86] et a conduit à de nombreuses réalisations depuis.

Dans [Leve 90a], il est présenté une version 16 bits (HYETI 2) d'un tel microprocesseur, disposant de capacités de test en ligne au niveau programme, ainsi qu'une version 32 bits (HSURF32), disposant des mêmes capacités au niveau programme, avec en plus, une implantation de son contrôleur suivant la méthode de S-codage décrite en section 1.5 . Par ailleurs, ce microprocesseur a été implanté avec des outils CAO de type "compilateur de silicium", ce qui a permis d'obtenir facilement plusieurs déclinaisons de ce circuit avec différentes possibilités de test en ligne, ceci afin de pouvoir chiffrer exactement le surcoût entraîné par l'introduction de ces dispositifs [Leve 90c].

Une version améliorée du processeur décrit dans [Leve 90a] a été réalisée dans le cadre d'un projet d'étudiants [Mich 92]. Sa particularité par rapport à la version de [Leve 90a] est son contrôleur dont la conception a été complètement repensée en vue du S-codage, ceci permettant de réduire au minimum le surcoût entraîné par la transformation du graphe. Par ailleurs, la génération de la signature pour le test au niveau programme a été légèrement modifiée afin d'améliorer l'efficacité de la vérification.

1.6.2. Test au niveau programme avec moniteur intégré au processeur

Si les études de méthodes ESM ont été assez nombreuses, l'étude de leur coût d'implantation, par contre a été nettement moins souvent menée, mis à part dans quelques cas où le moniteur est implanté en composants discrets [Schu 87]. La plupart des auteurs s'accordent cependant à dire qu'une intégration du moniteur

au processeur vérifié permet de réduire considérablement le coût matériel. Cette solution a également l'avantage de ne pas nécessiter de composant supplémentaire, ce qui est bénéfique pour la fiabilité générale du système.

Avec un moniteur intégré au processeur, il faut cependant faire attention aux hypothèses de fautes pour la détection par analyse de signature car il existe des modes de défaillance communs au processeur et au moniteur.

1.6.2.1. Génération d'une signature dérivée

L'avantage d'un moniteur intégré au processeur vérifié ne se situe pas seulement au niveau du coût matériel. L'observabilité du processeur est la caractéristique essentielle de ce type d'implantation[1]. Cela signifie que la représentativité de la signature par rapport au flot de contrôle du processeur vérifié peut être meilleure que dans le cas d'une génération de signature externe. Dans HSURF32, cette possibilité est exploitée de deux manières différentes :

- possibilité de définir le type des informations compactées, et ceci par simple programmation d'un registre du processeur,
- possibilité de compacter des informations de différentes manières, de telle sorte que la signature soit représentative à la fois de la valeur et du type de l'information compactée.

Le dispositif de génération de signature de HSURF (commun à toutes les versions) est présenté sur la figure 1.19 . Il comprend :

- un MISR contenant la signature courante (SIGN),
- le polynôme diviseur du MISR (POL) programmable,
- un dispositif de contrôle des informations compactées (RTIS et AS).

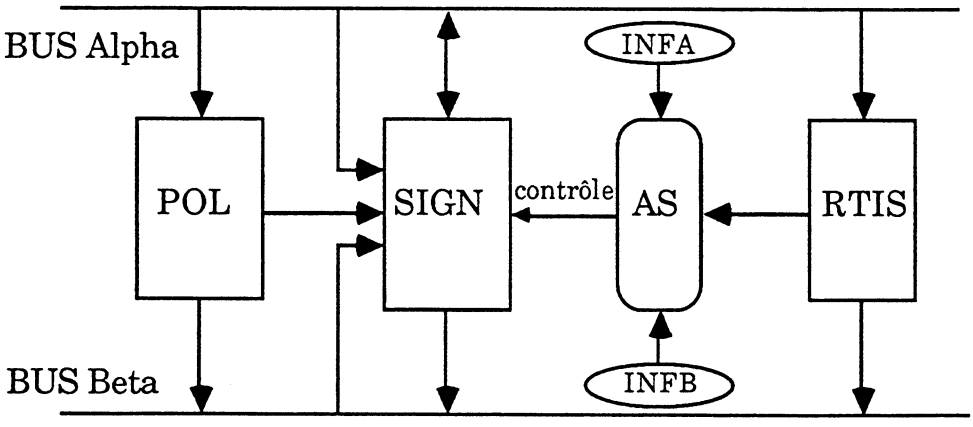


Figure 1.19 : dispositif de génération de signature de HSURF.

---

[1] C'est l'observabilité du processeur qui permet, entre autres, de réduire le coût matériel.

Les registres SIGN, POL et RTIS peuvent être accédés comme n'importe quel registre du microprocesseur.

Les signaux "INFA" et "INFB" sont émis par le contrôleur du circuit. Ils indiquent, à chaque cycle d'horloge, le type de l'information présente sur chaque bus. Le dispositif AS compare alors ce type avec le contenu du registre RTIS. Si le type de l'information présente sur l'un des bus est à compacter, le MISR est activé par le bloc de commande AS. En cas de conflit, seule l'information sur le BUS Beta est prise en compte.

Pour améliorer la souplesse du fonctionnement de l'analyseur, la nouvelle version de HSURF32 intègre un mécanisme permettant de modifier certains bits du registre RTIS à chaque instruction (Figure 1.20). Dans le code opératoire d'une instruction, certains bits sont destinés au contrôle du générateur de signature et sont chargés dans le registre RTIS au moment du chargement du registre de code instruction. Les bits 4, 5 et 6 du registre RTIS peuvent donc être différents pour chaque instruction. Ceci permet de compacter facilement et sans perte de temps certaines constantes de l'exécution du programme.

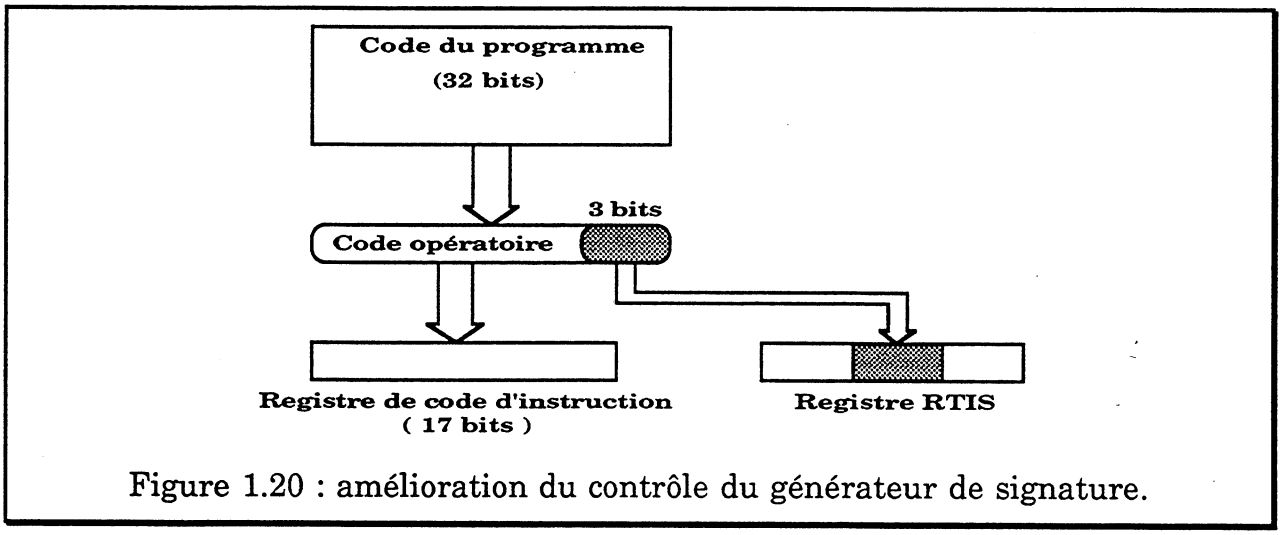


Figure 1.20 : amélioration du contrôle du générateur de signature.

Une seconde amélioration du générateur de signature consiste à compacter des données différemment suivant leur type.

Ceci est fait dans HSURF32 en différenciant les codes opératoires, les données immédiates et les adresses immédiates. Ces trois types d'informations sont ceux présents dans le code exécutable par le processeur. L'intérêt de compacter différemment chaque type est résumé sur la figure 1.21 .

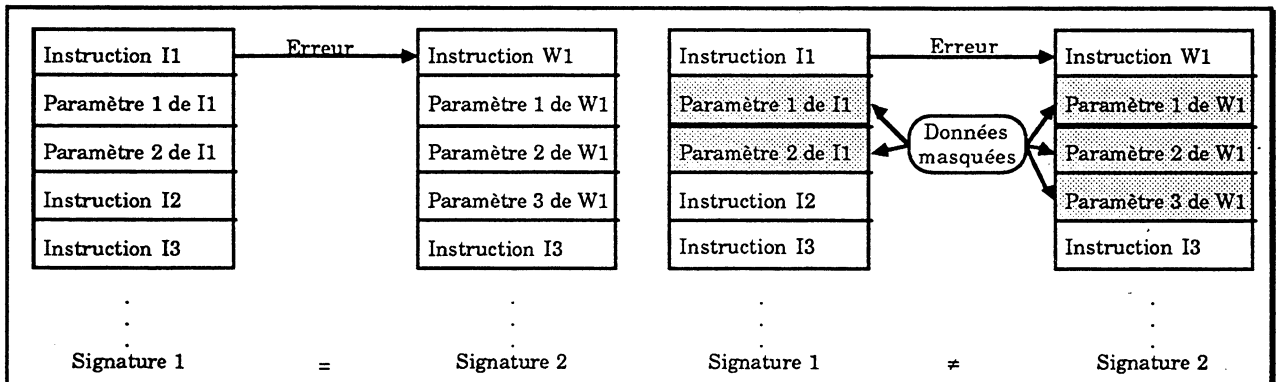
Supposons qu'une erreur du processeur modifie le format de l'instruction en cours d'exécution (erreur de décodage, par exemple). Dans un tel cas, suivant que le nombre de paramètres effectivement lu est supérieur ou inférieur au nombre de

paramètres normal, des codes opératoires vont être lus comme des paramètres ou des paramètres vont être exécutés comme instructions. Dans les deux cas, le processeur retrouvera son flot d'instructions normal au bout de quelques instructions sauf si une instruction de séquencement est touchée.

Si le séquencement du processeur n'est pas affecté suite à une erreur de format, la suite des mots lus dans le code est séquentielle, donc tout à fait normale. Pourtant, des instructions auront pu être "sautées" (cas de la figure 1.21) ou des instructions aléatoires auront été exécutées.

Avec une compaction de tous les mots du code du programme, sans distinction de leur type, la signature ne sera pas affectée par l'erreur de format (Figure 1.21a). La signature ainsi calculée n'est donc pas complètement représentative du flot de contrôle réel du processeur.

Pour remédier à ce cas de masquage, la solution consiste à compacter les mots lus dans le code en tenant compte de leur type (Figure 1.21b). Ainsi la signature est représentative de la valeur des mots lus dans le code du programme mais aussi de leur type, tel qu'il est vu par le microprocesseur pendant la lecture du code. Une erreur de format fausse donc la signature, même si la suite des mots lus dans le code est normale (séquentielle).



(a) compaction classique (b) compaction avec typage

Figure 1.21 : prise en compte du type des informations compactées.

Dans le cas de HSURF32, le type d'une instruction correspond à un "masque", qui inverse simplement certains bits de la donnée compactée. Ainsi une erreur de format est elle transformée en erreur de bit, ce qui peut être détecté à travers la signature.

Environ 30% des erreurs de décodage peuvent être détectées dans HSURF32, grâce à la compaction avec typage, alors quelles seraient totalement masquées avec une compaction normale. Sachant que sur les erreurs de décodage de HSURF32,

environ 30% perturbent le séquencement du programme et 10% sont détectées comme codes invalides, la détection des erreurs de décodage est donc assurée globalement à près de 70% (soit une amélioration de 75% grâce à la compaction avec typage). Dans [Mich 92b], des résultats similaires sont obtenus pour le cas d'un microprocesseur standard (Intel 8086), bien que les formats d'instructions soient très différents.

1.6.2.2. Invariance et test de la signature

Le schéma d'invariance de HSURF consiste à placer des ajustements sur certains noeuds du programme vérifié (cf 1.4.4.2.2).

Les ajustements et les tests de signature sont alors effectués par des instructions spécifiques, prévues dans le jeu d'instruction du microprocesseur, et exécutées comme toutes les autres instructions. Les valeurs associées aux singularités sont les opérandes de ces instructions et peuvent être adressées par l'ensemble des modes d'adressage du microprocesseur (il en existe 16).

Pour les sous-programmes, la technique utilisée est identique à celle décrite dans [Sosn 88] (cf 1.4.4.2.3), c'est-à-dire une technique privilégiant, dans le cas d'un moniteur intégré, la rapidité d'exécution pour les départs et retours de sous-programmes.

Pour les départs en exceptions, la signature est empilée dans la pile du processeur, en même temps que les autres informations (compteur ordinal, registre d'état, ...).

1.6.3. Test au niveau graphe d'état du contrôleur

La modification du graphe de contrôle de la version originale de HSURF32 [Leve 90a] avait conduit à une augmentation de la surface du circuit de l'ordre de 25%, ce qui est tout à fait inacceptable. Le contrôleur, seul, avait pour sa part augmenté d'environ 50%.

Ceci a conduit à redéfinir le graphe de contrôle de HSURF32 en prenant en compte les contraintes du S-codage dès le début de la conception.

La méthode utilisée a consisté dans une première étape à supprimer une partie des arcs rebouclant sur leur état de départ. En effet, une boucle sur un état impose des contraintes très fortes au niveau de la S-codabilité et leur suppression favorise donc le S-codage. Les états d'attente ainsi supprimés ont été remplacés par un mécanisme externe de blocage des phases du contrôleur.

Après quoi, les états rebouclants dont les boucles n'ont pu être supprimées ont été "isolés" (Etats E31 et E2r). Enfin, les dernières contraintes de S-codage ont été résolues en répétant certains états. Les états E2, E7, E7b, E11, E12 et E19 ont ainsi été

répliqués. L'état E2 qui correspond au point de jonction de la plupart des chemins du graphe a du être répété cinq fois. Le graphe ainsi obtenu est présenté sur la figure 1.22 .

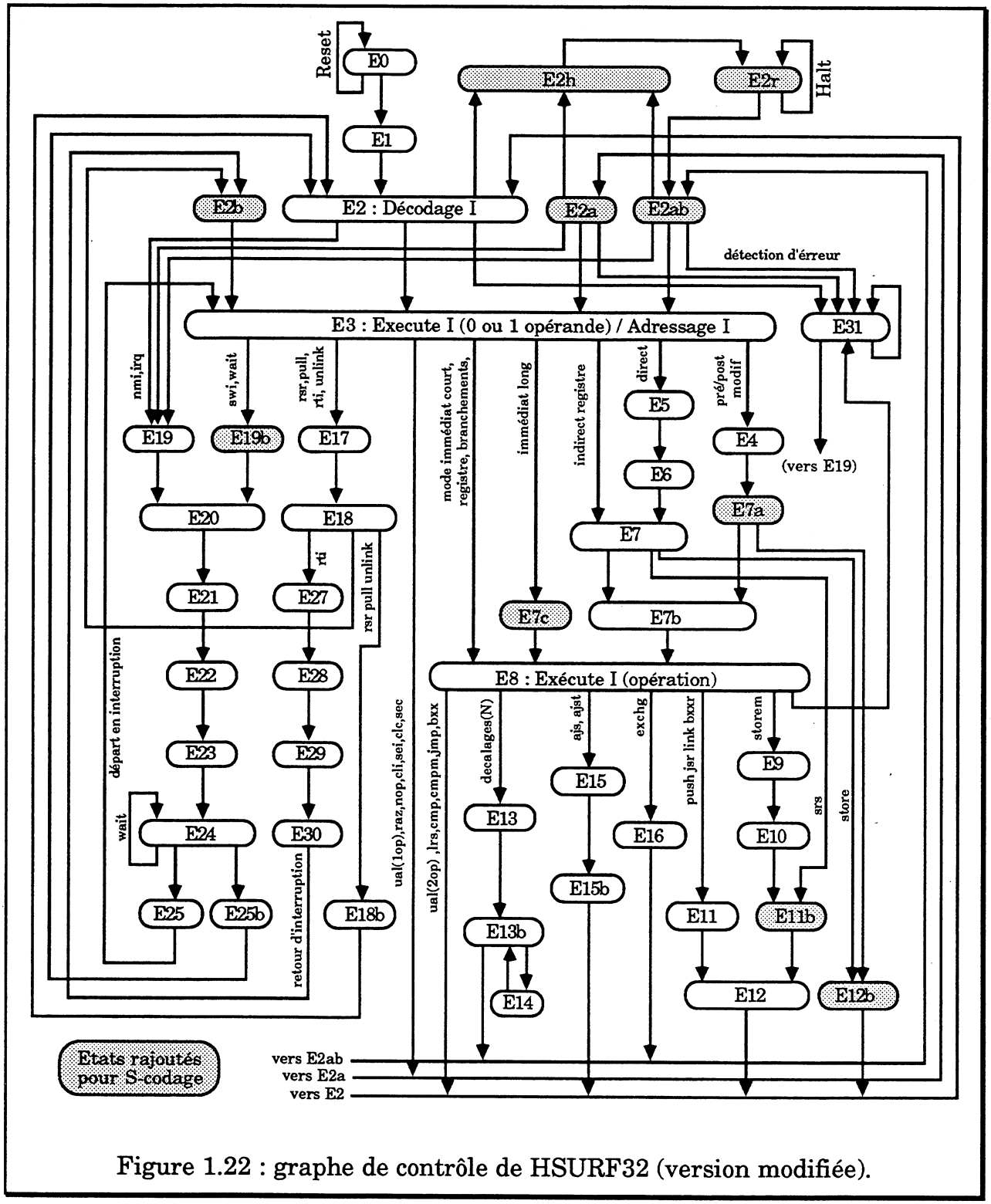


Figure 1.22 : graphe de contrôle de HSURF32 (version modifiée).

Tous les états répliqués pour le S-codage ont été choisis parmi les états générant peu d'équations de commande. L'impact du S-codage sur la logique du contrôleur est donc très faible (environ 1% d'augmentation).

Pour la vérification de la signature du contrôleur, l'état E3 du graphe a été choisi comme seul point de test. La signature est donc vérifiée avant le début de l'exécution de chaque nouvelle instruction.

Les dispositifs matériels introduits pour la vérification du contrôleur sont présentés sur la figure 1.23 et correspondent à une implantation tout à fait typique de contrôleur S-codé. La comparaison est effectuée grâce à un comparateur câblé (OU ayant autant d'entrées que le MISR) ce qui permet de simplifier la logique de vérification et d'accélérer la comparaison. Le signal d'erreur ainsi obtenu est reporté directement sur un des plots du circuit pour être traité par un dispositif externe. En effet, contrairement aux autres sources d'erreurs (codes opératoires illégaux, erreur sur la signature du programme exécuté), le traitement d'une erreur de séquencement interne par "trap" n'est pas envisageable car le microprocesseur peut ne plus être en état d'exécuter une instruction.

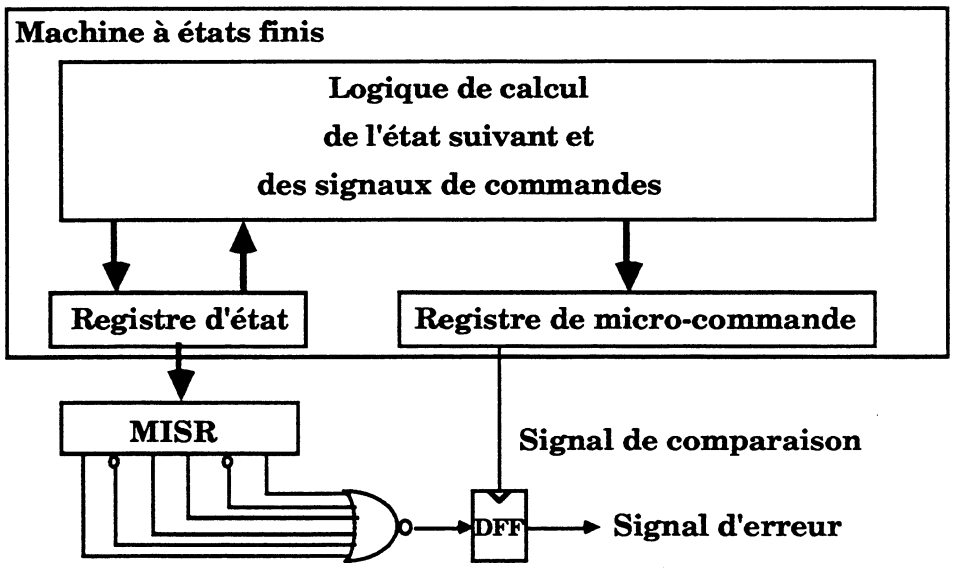


Figure 1.23 : implantation d'un contrôleur S-codé (HSURF32).

1.6.4. Implantation du circuit

Le circuit se décompose en une partie opérative et une partie contrôle :

La partie opérative est constituée pour l'essentiel d'un bloc de 32 bits implanté avec un générateur automatique de chemin de données en tranches (outils COMPASS). Une petite partie en cellules standard contient le registre d'état du microprocesseur, le dispositif de contrôle de l'analyseur de signature au niveau programme, un registre de mémorisation et de contrôle des sources d'erreur et les amplificateurs des plots du bus de données bidirectionnel.

La partie contrôle est implantée sous forme de trois zones de cellules standard :

- une zone contenant la logique du contrôleur S-codé, le registre d'état du contrôleur, le dispositif de vérification du séquencement du contrôleur et la logique d'interface avec le bus de contrôle du microprocesseur,
- une zone contenant le registre de micro-instruction,
- une zone contenant le décodage des champs opérande (registres) d'une instruction.

L'architecture du circuit complet est présentée sur la figure 1.24.

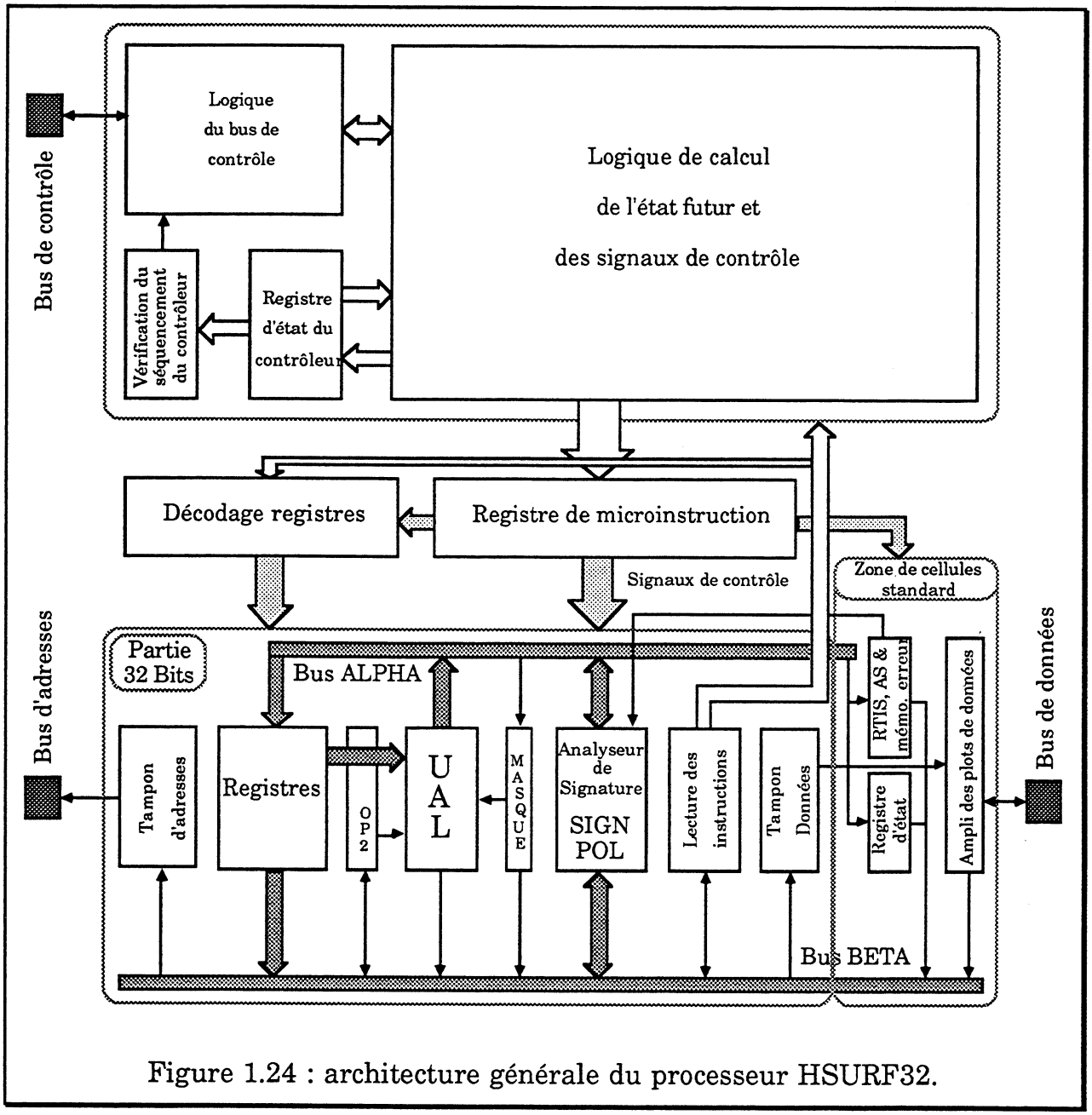


Figure 1.24 : architecture générale du processeur HSURF32.

1.7. Conclusion

Dans ce chapitre, différentes méthodes de vérification du flot de contrôle d'un système ont été décrites. La plupart de ces méthodes sont destinées à la vérification au niveau programme et utilisent une signature dérivée, générée à partir des instructions exécutées par le processeur du système vérifié. Parmi ces méthodes, celles qui ont été le plus étudiées sont les méthodes ESM car elles présentent un coût d'implantation généralement très faible. Pour ces méthodes, l'intégration du moniteur au processeur vérifié, comme dans le cas de HSURF, est la solution d'implantation idéale, d'un point de vue efficacité, coût matériel et facilité de mise en oeuvre (accès au compilateur). Cependant, la réalisation d'un processeur spécifique n'est pas toujours envisageable.

Pour l'adaptation d'une méthode de vérification de flot de contrôle dans un système industriel, l'introduction des références dans le code, pour une méthode ESM, est un problème très délicat (difficulté d'accès aux compilateurs). Une méthode DSM, dont l'avantage est de préserver la compatibilité logicielle, peut donc être préférable pour un tel système, malgré son coût d'implantation matériel a priori nettement plus élevé.

Cependant, ces méthodes DSM n'ont reçu jusqu'ici que peu d'intérêt et les rares méthodes proposées, toutes très différentes, ne sont pas totalement satisfaisantes. En particulier, dans les méthodes proposées jusqu'à maintenant, le test de la signature n'est jamais fait par comparaison avec une référence unique, précalculée et mémorisée comme telle, contrairement aux méthodes ESM. Toutes ces méthodes présentent par ailleurs des coûts d'implantation très élevés.

Les chapitres 2 et 3 sont donc consacrés à la description de deux nouvelles méthodes DSM dont le but est de présenter une efficacité au moins comparable aux méthodes ESM tout en minimisant le coût matériel du dispositif de vérification et le coût mémoire des informations pour le test.

La méthode présentée au chapitre 2 est prévue pour être d'un usage aussi général que possible et s'applique facilement à différents microprocesseurs standards. En particulier, elle s'affranchit du décodage des instructions du processeur pour le repérage des singularités, mais sans pour autant nécessiter de mémoire étiquette. Par ailleurs, elle est utilisable sans dispositif annexe de réduction de la latence de détection, contrairement à la plupart des méthodes ESM qui imposent de vérifier que la signature est bien testée régulièrement.

La méthode présentée au chapitre 3 est dérivée de la précédente dans le but de diminuer le coût matériel dans le cas particulier de l'implantation qui sera décrite

au chapitre 4. Cette méthode utilise une technique d'ajustements sur arcs pour assurer l'invariance de la signature. L'utilisation d'ajustements pour une méthode DSM n'avait encore jamais été proposée. On verra que ceci permet, comme dans le cas d'une méthode ESM, de réduire sensiblement le coût mémoire des informations de test, au moins dans certains cas.

Il sera également montré, sur un exemple de système, au chapitre 4, que les techniques de vérification au niveau graphe d'états peuvent avoir des applications au niveau programme pour la vérification, par signature dérivée, du séquencement d'un processeur.

Chapitre 2. WDP : une méthode DSM modulaire

2.1. Introduction

La vérification du flot de contrôle d'un microprocesseur, par signature dérivée, est une technique de test en ligne efficace mais l'implantation d'une méthode ESM se heurte, dans certains cas, au problème de l'introduction des références dans le code du programme vérifié. Une méthode DSM peut donc, dans ces cas là, être préférable.

La méthode WDP (Watchdog Direct Processing) est donc une méthode DSM qui a été conçue pour essayer de répondre à un double objectif : pouvoir offrir une méthode de vérification d'un flot de contrôle, par analyse de signature dérivée, aussi générale que possible, tout en arrivant à minimiser à la fois le coût de mise en œuvre, le coût mémoire, le coût matériel et la latence de détection.

En ce qui concerne le coût d'implantation a priori élevé d'une méthode DSM, il faut cependant constater deux points :

- La possibilité d'employer une mémoire séparée pour les références fait que l'on peut utiliser, dans certains cas, des circuits mémoires moins rapides que ceux utilisés pour la mémoire des instructions du processeur, étant donné que les accès y sont moins fréquents. Compte tenu des différences de prix et de fiabilité énormes qui existent entre les mémoires les plus rapides et d'autres plus lentes, la solution la plus économique n'est pas forcément celle qui propose le surcoût mémoire le plus faible, au moins pour les processeurs ayant une mémoire d'instruction à accès très rapide.
- Avec la complexité croissante des processeurs usuels, un analyseur de signature utilisant des références internes au programme devient également très complexe [Delo 91]. L'avantage des méthodes ESM au niveau coût matériel s'atténue donc avec la complexité du processeur vérifié. Par ailleurs, les compilateurs de ces microprocesseurs sont également délicats à modifier de façon totalement fiable.

On peut donc remarquer que pour les processeurs les plus performants, une méthode DSM peut se justifier, même en terme de complexité matérielle, par rapport à une approche ESM. Quand on ajoute à cela le respect de la compatibilité logicielle, on s'aperçoit qu'une méthode DSM est particulièrement bien adaptée à toute une génération de processeurs CISC dont l'existence même est basée sur la compatibilité logicielle. La méthode WDP décrite dans ce chapitre a été conçue suite à ces remarques.

2.2. Description de la méthode

2.2.1. Principe et origine de la méthode

2.2.1.1. Repérage des singularités par leur adresse

La première idée à l'origine de la méthode WDP consiste à repérer les singularités du programme par leur adresse, ce qui n'avait jamais été fait dans aucune autre méthode[1].

Les informations associées aux singularités sont, outre leur adresse dans le programme vérifié, le type de la singularité et une valeur permettant d'assurer l'analyse de la signature. Ces valeurs sont stockées dans une mémoire séparée du programme et forment un programme de vérification, accédé par un pointeur et exécuté par un watchdog. Le type de la singularité correspond au code opératoire d'une instruction pour le watchdog.

Le premier avantage de ce type de repérage est de ne pas nécessiter le décodage des instructions du processeur vérifié. Il peut donc s'appliquer d'une manière générale, quels que soient le processeur et son format d'instruction.

2.2.1.2. Vérification directe du séquencement du processeur

Pour la gestion du pointeur sur le programme de vérification, le watchdog doit imiter le séquencement du processeur vérifié. Pour cela, il doit disposer, dans son programme de vérification, d'informations sur la structure du programme vérifié. Les autres méthodes DSM procèdent également de cette manière mis à part OSLC qui n'utilise pas de pointeur.

La deuxième idée de base de WDP consiste, pour assurer à la fois la gestion du pointeur du watchdog et une détection efficace des erreurs de séquencement du processeur, à placer une instruction watchdog en correspondance avec chaque instruction à l'origine ou à la destination d'une rupture de séquence du programme vérifié. Les singularités, dans la méthode WDP, sont donc toutes des extrémités de blocs linéaires (cf 1.4.1).

Ainsi le watchdog connaît, par un moyen externe, l'emplacement de toutes les instructions de séquencement dans le programme vérifié et il peut donc détecter les ruptures de séquence inopinées. De plus, l'adresse de la destination d'une

---

1 L'association d'une singularité et de son adresse a déjà été proposée, dans la méthode OSLC (cf 1.4.5.3), mais à des fins de test et non pas de repérage.

instruction de séquencement étant connue par le watchdog, celui-ci peut également détecter les branchements erronés. Ceci est donc un deuxième avantage de l'utilisation du repérage des singularités par leur adresse, dans le cas où, comme dans la méthode WDP, elles correspondent à des extrémités de blocs linéaires.

2.2.1.3. Décorrélation des signatures intermédiaires

La signature, de type dérivée, n'est utilisée dans WDP que pour la détection des erreurs de bits. La vérification de cette signature est effectuée en affectant une signature de référence à chaque singularité, l'invariance étant assurée par réinitialisation du dispositif de compaction à chaque branchement et non par ajustements.

La troisième idée de la méthode WDP consiste à ne pas réinitialiser la signature avec une valeur fixe en début de bloc linéaire, contrairement aux autres méthodes sans ajustements. On a vu en section 1.4.4.1.4 que ceci entraîne une corrélation des signatures intermédiaires néfaste à l'efficacité du test. Dans WDP, la signature est réinitialisée en cas de rupture de séquence avec la valeur de référence associée à la destination. En cas d'arrivée séquentielle sur une destination (point de jonction), la signature est alors testée avec la signature de référence mais elle n'est pas réinitialisée. De cette manière, la signature à un point de jonction est invariante mais avec une valeur différente pour chaque point de jonction, ce qui permet d'obtenir une répartition aléatoire des signatures intermédiaires sur l'ensemble du programme.

2.2.2. Constitution du programme de vérification

Dans la suite de ce chapitre, les instructions du watchdog seront appelées "noeuds". Il en existe deux sortes, suivant le type de l'instruction correspondant à la singularité.

Définition 2.1 : un noeud de séquencement est une instruction du watchdog correspondant à une instruction de séquencement.

Définition 2.2 : un noeud destination est une instruction du watchdog correspondant à une destination.

Un même noeud peut être à la fois séquencement et destination dans le cas d'un bloc linéaire dégénéré (cf 1.4.1, figure 1.4.b). Ce cas correspond à une instruction de séquencement, tenant sur un seul mot mémoire du programme, et qui se trouve être une destination (cas peu fréquent).

2.2.2.1. Informations relatives à un noeud

Les informations relatives à un noeud sont les suivantes :

- Champ 1 : type de l'instruction du programme vérifié correspondant au noeud,
- Champ 2 : localisation de ce noeud dans le programme vérifié (adresse dans le programme vérifié),
- Champ 3 : valeur de référence de la signature à ce noeud,

et dans le cas d'un noeud de séquencement :

- Champ 4 : adresse dans le programme watchdog du noeud destination.

2.2.2.2. Utilisation d'une technique de hachage

Une instruction watchdog devrait donc contenir trois ou quatre champs selon qu'il s'agit d'un noeud destination ou d'un noeud séquencement. Afin d'avoir un format d'instruction fixe, et surtout de réduire l'encombrement mémoire du programme watchdog, les champs 3 et 4 sont fusionnés, dans le cas des noeuds de séquencement, par l'emploi d'une technique de hachage : le champ 3 d'un noeud est remplacé par le OU Exclusif des champs 3 et 4.

Sur un noeud de séquencement, le calcul de l'adresse (dans le programme watchdog) du noeud destination consiste donc à faire un hachage inverse du champ 3 avec la signature courante.

Soit PCW le pointeur d'instruction du watchdog. Sur un noeud de séquencement (pointé par PCW_{actuel}), le PCW destination (PCW_{dest}) doit correspondre au noeud associé à l'instruction destination de l'instruction de séquencement et il sera calculé de la manière suivante :

$$PCW_{dest} \leftarrow SIGN \oplus REF(PCW_{actuel})$$

PCW_{actuel}	pointe un noeud de séquencement
SIGN	est la signature au noeud pointé par PCW_{actuel}
$REF(PCW_{actuel})$	est le champ 3 (référence) de l'instruction watchdog pointée par PCW_{actuel}
PCW_{dest}	est l'adresse dans le programme watchdog du noeud destination de l'instruction correspondant au noeud pointé par PCW_{actuel}

De cette manière, les signatures à un noeud de séquencement et la destination de l'instruction de séquencement sont vérifiées avec un seul champ dans le programme watchdog.

En effet, si la signature est erronée à une instruction de branchement, la destination calculée par le watchdog sera faussée (erreur de séquencement du watchdog). En cas d'erreur de séquencement du processeur vérifié ou du watchdog, la détection est assurée par le fait que l'adresse du branchement ne correspondra pas à celle trouvée par le watchdog dans le programme de vérification.

Un cas de non détection peut toutefois se produire dans le cas d'une erreur de séquencement simultanée des deux processeurs (signature erronée à un branchement erroné). La probabilité de ce masquage d'erreur est très faible et sera détaillée en section 2.4.3 .

La technique de hachage utilisée pour WDP est très proche de celle utilisée dans SIS (cf 1.4.4.1.2). La différence vient du fait que, pour WDP, c'est le séquencement du processeur watchdog qui est affecté par une erreur de signature et non pas celui du processeur vérifié. Combiné à un repérage externe pour les extrémités de blocs, qui permet la détection des erreurs de séquencement, la probabilité de masquage entraînée par le hachage dans WDP est largement plus faible que dans SIS.

Cependant, cette technique de hachage ne présente pas que des avantages.

Avantages du hachage :

- Toutes les instructions watchdog ont le même format (3 champs).
- Réduction de la taille mémoire du programme watchdog.

Inconvénients du hachage :

- Une instruction watchdog unique ne peut représenter une instruction de séquencement qui est aussi une destination (bloc linéaire dégénéré, cf figure 1.4b), car le champ 3 (référence) n'a pas la même signification dans le cas d'un noeud séquencement et dans le cas d'un noeud destination. Il est donc impératif de faire correspondre deux instructions watchdog à ces instructions particulières ce qui nécessite quatre mots de références pour ce type de singularité alors que trois suffirait sans le hachage des références. Si plus de 50% des instructions de séquencement sont aussi des destinations (cas exceptionnel), la réduction du coût mémoire mentionnée comme avantage du hachage peut donc s'inverser.

- La vérification de la signature n'est pas assurée dans le cas d'un branchement conditionnel non pris. Ceci peut augmenter la latence de détection des erreurs de bit, particulièrement dans le cas où plusieurs noeuds de ce type sont consécutifs.

- Le hachage impose d'avoir une bijection entre l'ensemble des signatures et l'espace d'adressage du watchdog. Pour une référence donnée stockée dans le champ 3, le hachage avec une signature donnée doit fournir une adresse et une seule (injection), différente de toutes celles pouvant être obtenues avec d'autres signatures (surjection).
    - La propriété d'injection est assurée par le fait que la fonction "OU Exclusif" est appliquée bit à bit, toujours sur des bits de même poids.
    - Une condition suffisante pour que la fonction de hachage soit surjective est de calculer des signatures d'une largeur au moins égale à celle du bus d'adresses du processeur watchdog.

2.2.2.3. Localisation des noeuds avec un champ réduit

Au niveau du champ localisation des instructions watchdog, plutôt que de stocker l'adresse microprocesseur complète du noeud, on peut se contenter de ne conserver qu'un sous-ensemble des bits de cette adresse. Nous verrons en section 2.6.1 que ceci est indispensable pour vérifier le code d'un processeur disposant d'une MMU intégrée.

Dans tous les cas et même si ce n'est pas indispensable, la réduction du champ de localisation permet de diminuer le coût mémoire (en nombre de bits) du programme watchdog. Ceci permet par la même occasion de placer les champs 1 et 2 d'un noeud sur un seul mot mémoire dans le programme watchdog.

Par exemple, pour un format d'instruction watchdog sur 16 bits, le champ 1 "Type du noeud" peut être placé sur 4 bits et le champ 2 "localisation" sur 12 bits (adresse réduite). Ainsi, avec une signature sur 16 bits également, chaque noeud du programme watchdog tient sur deux mots de 16 bits.

La localisation par adresses réduites présente toutefois des inconvénients relatifs aux possibilités de masquage supplémentaires :

- En cas de branchement erroné dont les bits de poids faible sont corrects, une erreur de séquencement ne sera pas détectée immédiatement. Elle pourra être détectée car le watchdog et le microprocesseur seront décalés, mais avec une certaine latence. Une telle erreur peut ne pas être détectée si le watchdog et le microprocesseur se rejoignent par des chemins différents mais équivalents.
- En cas de branchement, seuls les poids faibles de l'adresse destination sont vérifiés et les poids forts doivent donc être réinitialisés avec l'adresse utilisée par le microprocesseur, sans possibilité de détection immédiate dans le cas où ces poids forts seraient erronés.

L'analyse et le chiffrage de ces possibilités de masquage seront présentés en section 2.4 . En fait, l'utilisation d'adresses réduites pour la localisation des noeuds modifie moins les taux de couverture d'erreurs que la latence de détection de ces erreurs.

Il faut remarquer que, même dans le cas d'une localisation par adresses réduites, la détection des branchements du processeur vérifié doit toujours se faire sur l'ensemble des bits de l'adresse du microprocesseur. En effet, il peut arriver que certains branchements ne modifient que les bits de poids fort de l'adresse et dans ce cas, une détection de branchement sur les poids faibles uniquement serait prise en défaut. L'utilisation d'adresses réduites pour la localisation ne permet donc pas une réduction de la complexité du watchdog mais seulement une réduction du coût mémoire du programme de vérification.

2.2.2.4. Structure des instructions watchdog

Les instructions pour le watchdog WDP ont toutes le même format, mais suivant le type du noeud, le champ de référence correspond à une signature de référence ou à une référence hachée. La figure 2.1 présente un exemple de format d'instructions WDP dans le cas d'une signature sur 16 bits et une localisation des singularités par les 12 bits de poids faible de leur adresse. Le champ "Type" d'une instruction est indépendant du format et tient sur 4 bits au maximum en considérant toutes les instructions de séquencement classiques des microprocesseurs usuels.

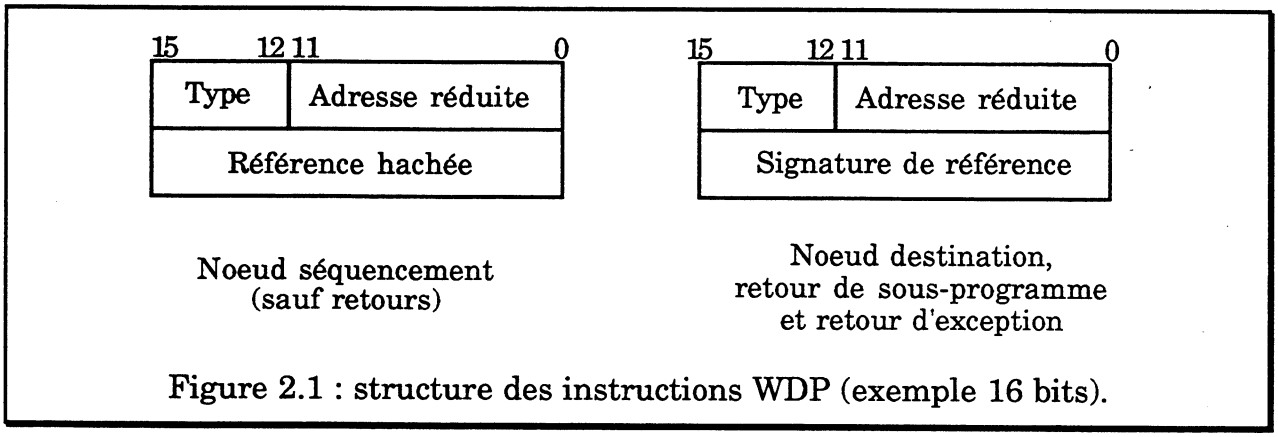


Figure 2.1 : structure des instructions WDP (exemple 16 bits).

Grâce à l'utilisation d'une technique de hachage et à une localisation par adresses réduites, toutes les instructions WDP ont le même format et tiennent sur seulement deux mots mémoire. Ceci facilite la lecture anticipée des instructions du watchdog car il n'est pas nécessaire de décoder une instruction pour connaître sa taille puisque toutes ont le même format.

En ce qui concerne le champ de localisation des singularités, il faut noter que sa signification n'est pas la même suivant le type du noeud :

- pour un noeud de séquencement (avec référence hachée ou référence explicite), l'adresse de localisation du noeud correspond au dernier mot de l'instruction de séquencement, et non pas au code opératoire de cette instruction (sauf pour les instructions sur un seul mot),
- pour un noeud destination, l'adresse de localisation du noeud correspond au premier mot de l'instruction destination, donc au code opératoire.

En effet, une instruction de séquencement marque toujours la fin d'un bloc linéaire et l'adresse de localisation est donc celle du dernier mot devant être compacté avant réinitialisation de la signature, en cas de branchement.

Au contraire, une destination est toujours un début de bloc linéaire et l'adresse de localisation est donc celle du premier mot devant être compacté.

2.2.3. Fonctionnement du watchdog

Un exemple de situation de programme est représenté sur la figure 2.2 .

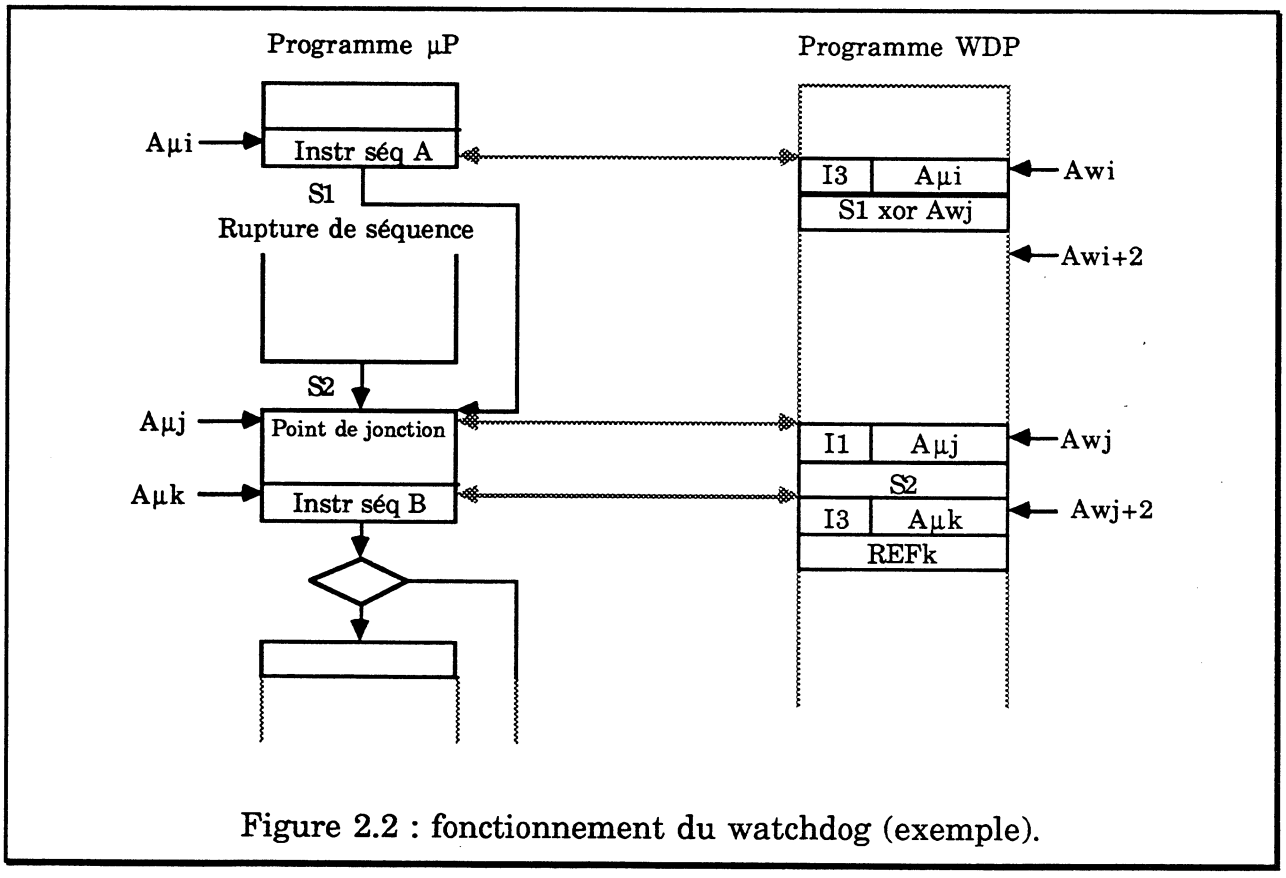


Figure 2.2 : fonctionnement du watchdog (exemple).

Les notations suivantes sont utilisées dans la figure 2.2 :

- Aµi : adresse de localisation du noeud i dans le programme microprocesseur (première adresse de l'instruction pour un noeud

destination et dernière adresse de l'instruction pour un noeud séquencement),

- A_{wi} : adresse dans le programme watchdog du noeud i.

Le fonctionnement du watchdog consiste en premier lieu à attendre que le microprocesseur arrive sur une singularité. Pour cela il compare les adresses du processeur avec le champ de localisation du noeud en cours de traitement. Lorsque le microprocesseur arrive sur une singularité, le watchdog traite la singularité.

2.2.3.1. Fonctionnement sur une instruction de séquencement

Lorsque le microprocesseur arrive sur une instruction de séquencement, le watchdog (noeud en cours : I3 @ Awi, sur la figure 2.2) calcul l'adresse du noeud destination (Awj) en utilisant la signature courante (S1) et le champ de référence haché du noeud traité. Puis il charge ce noeud destination (I1 @ Awj sur la figure 2.2).

Quand le processeur effectue le branchement, le watchdog compare l'adresse de la destination prise par le processeur et le champ de localisation du noeud qu'il vient de charger (I1 @ Awj). En cas de différence, une erreur est signalée. Puis la signature est réinitialisée avec le champ de référence du noeud destination (S2). Le watchdog charge enfin le noeud situé en séquence dans son programme de vérification (Awj + 2) et il attend que le microprocesseur arrive sur la singularité correspondant à ce noeud.

Si le processeur n'effectue pas le branchement (instruction conditionnelle) le watchdog charge alors simplement le noeud situé en séquence dans son programme de vérification (Awi + 2) et il attend que le microprocesseur arrive sur la singularité correspondant à ce noeud.

2.2.3.2. Fonctionnement sur un point de jonction

Lorsque le microprocesseur arrive séquentiellement sur une destination (point de jonction), le watchdog (noeud en cours : I1 @ Awj) compare simplement la signature courante avec le champ référence du noeud. En cas de différence, une erreur est signalée. Puis le watchdog charge le noeud situé en séquence dans son programme de vérification (Awj + 2) et il attend que le microprocesseur arrive sur la singularité correspondant à ce noeud.

2.3. Génération des références

Dans cette section, la génération des informations de test pour le watchdog de la méthode WDP est décrite. Afin de bien monter la correspondance entre un programme vérifié et le programme de vérification qui lui correspond, deux

exemples sont présentés en section 2.3.2 . Puis l'algorithme de calcul des références est décrit en section 2.3.3 . Dans la section 2.3.4, une optimisation des références est proposée de manière à réduire le coût mémoire d'un programme de vérification.

2.3.1. Exemple de programmes vérifiés

Dans l'exemple de la figure 2.3 , un exemple simple de programme vérifié est représenté (pas d'appels de sous-programmes). Dans le programme de vérification, il correspond une instruction watchdog à chaque noeud séquencement et à chaque noeud destination. Le programme du watchdog possède exactement la même structure que le programme vérifié.

Dans le cas des noeuds séquencement, l'adresse du noeud destination dans le programme watchdog n'est pas stockée explicitement mais elle est hachée avec la signature de référence du bloc linéaire précédant le noeud.

Dans le cas des noeuds destination, au contraire, l'instruction watchdog contient explicitement la référence du bloc linéaire précédant le noeud. Pour un noeud destination qui ne peut être atteint séquentiellement (destination située immédiatement après une instruction de séquencement inconditionnelle), la signature de référence ne sert en fait qu'à l'initialisation de la signature lors du branchement dont c'est la destination.

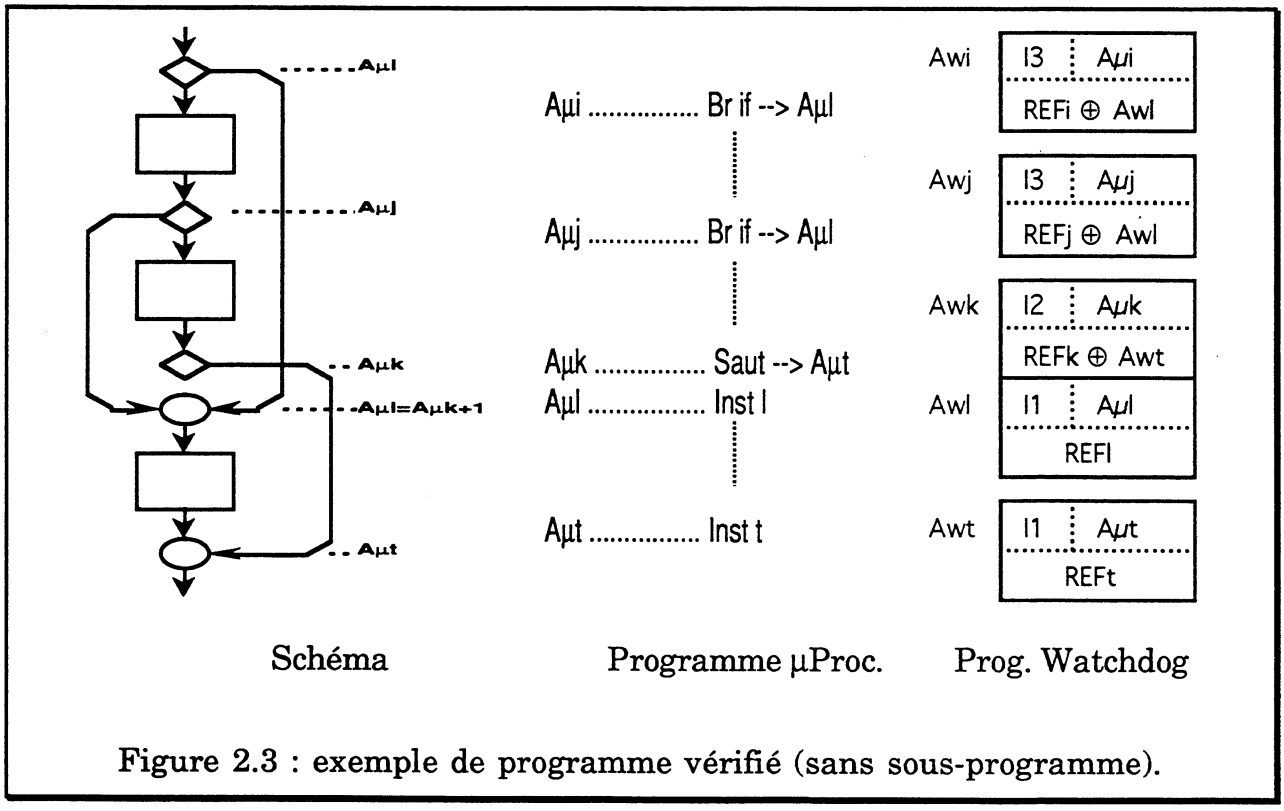


Figure 2.3 : exemple de programme vérifié (sans sous-programme).

Dans l'exemple de la figure 2.4 , la correspondance entre programme vérifié et programme de vérification est la même que dans l'exemple de la figure 2.3.

Cependant, le programme vérifié comporte un sous-programme appelé depuis deux endroits. Une instruction d'appel de sous-programme est un noeud séquencement normal, qui comporte, comme un branchement, une signature de référence hachée avec l'adresse du noeud destination. Par contre, une instruction de retour de sous-programme est un noeud séquencement particulier, car la signature de référence n'est pas hachée. En effet, l'adresse de retour dépend de l'endroit d'où le sous-programme a été appelé. L'adresse du noeud destination d'un retour n'est donc pas fixe et ne peut pas être hachée. La signature de référence du bloc est donc stockée explicitement.

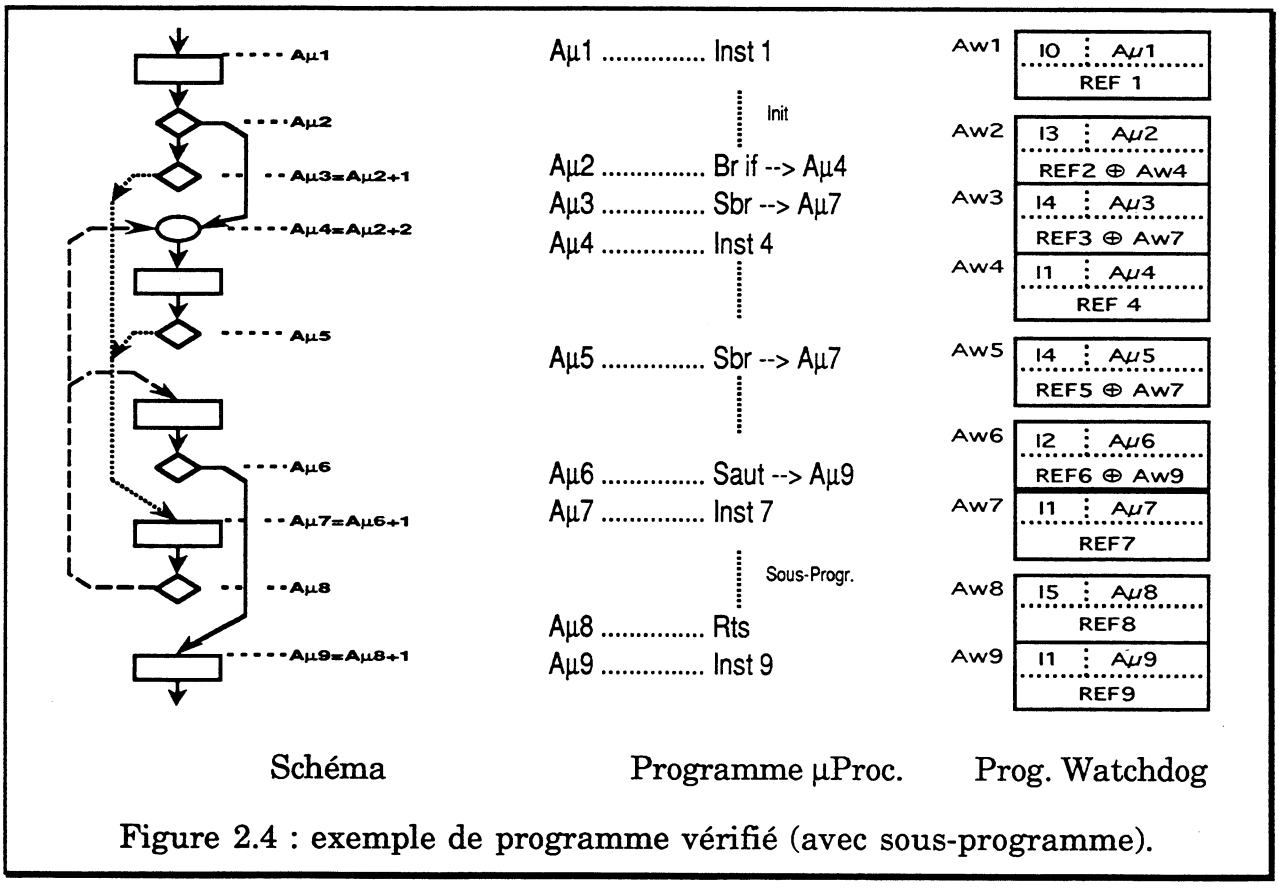


Figure 2.4 : exemple de programme vérifié (avec sous-programme).

2.3.2. Algorithme de création du programme de vérification

L'algorithme de génération du programme watchdog est très simple car la méthode WDP assure l'invariance de la signature par réinitialisation. Cet algorithme nécessite deux phases et s'exécute directement sur le code exécutable du programme vérifié. Pour chaque phase, le code du programme vérifié est parcouru linéairement et les instructions sont partiellement décodées (format et instructions

de séquencement). On suppose que le code du programme ne contient pas de sections de données ni de constantes, celles-ci devant être placées hors du code.

L'organigramme du programme de génération des informations du watchdog est décrit sur la figure 2.5 .

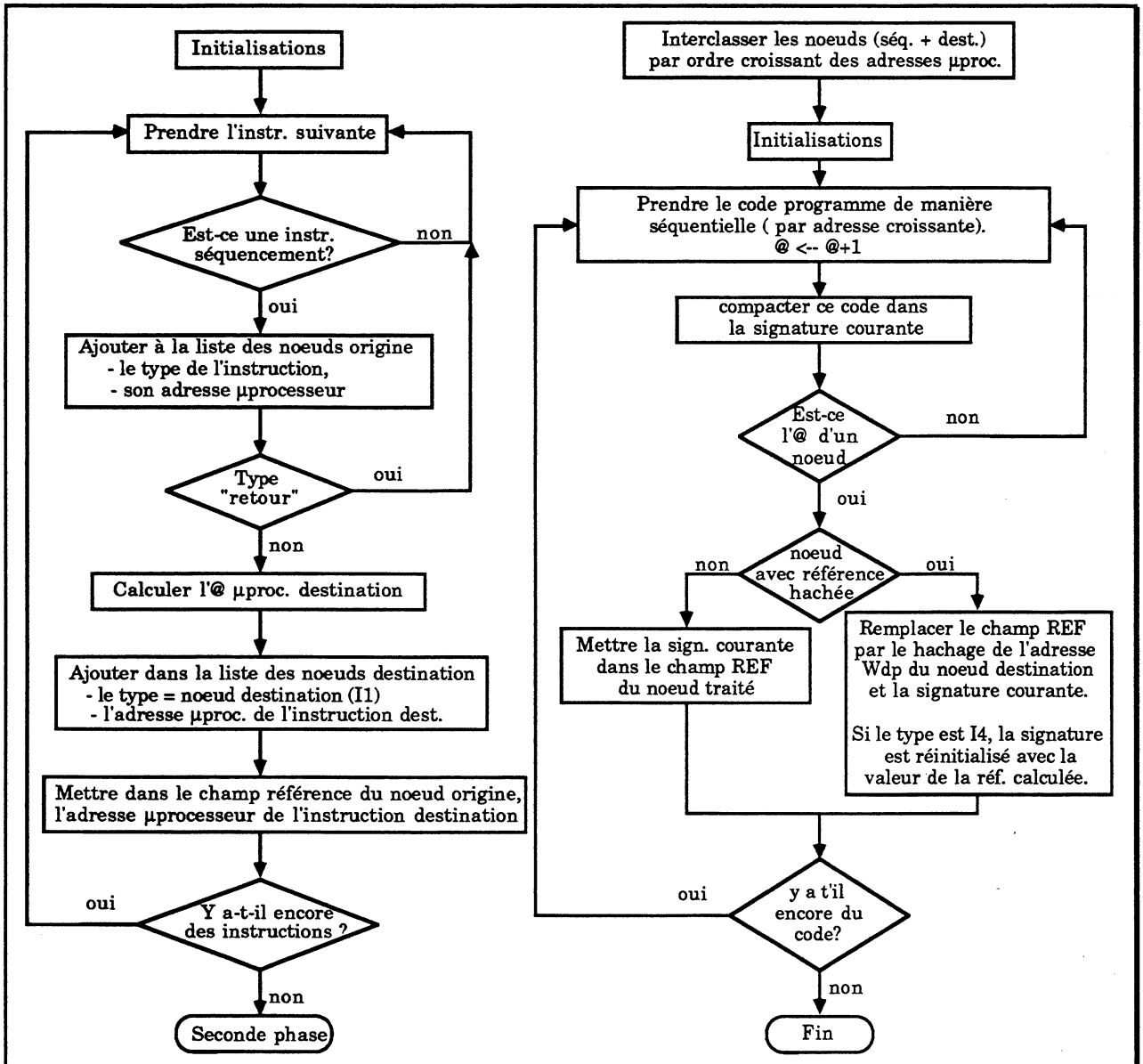


Figure 2.5 : algorithme de génération des programmes watchdog.

La première phase consiste à localiser toutes les singularités du programme, c'est-à-dire les instructions de séquencement et les destinations de ces instructions.

Après la phase 1, une étape de classement des noeuds par leur adresse de localisation est nécessaire.

La seconde phase consiste à calculer la signature intermédiaire en tout point du programme vérifié et à générer les informations définitives du programme de vérification. La répartition aléatoire des signatures intermédiaires est assurée en allouant aux noeuds ne pouvant pas être atteints de manière séquentielle, la valeur de la signature du bloc linéaire qui les précède séquentiellement (comme pour un noeud normal).

Dans la phase 2 de l'organigramme décrit sur la figure 2.5 , dans le cas d'un noeud avec référence hachée, le logiciel de génération du programme watchdog doit remplacer le champ référence du noeud courant par le hachage de la signature courante et de l'adresse watchdog du noeud destination. Cette adresse watchdog est obtenue grâce à l'adresse microprocesseur contenue dans le champ référence du noeud courant (destination de l'instruction de séquencement). L'adresse microprocesseur permet de rechercher l'adresse watchdog du noeud destination qui est sa position dans la liste des noeuds (à une base près).

Dans le cas d'un noeud de départ en sous-programme (I4), la signature est réinitialisée avec la valeur du champ référence (référence hachée). En effet, le noeud de départ est aussi la destination de l'instruction de retour (cf annexes 1.5).

Complexité :

Les données à traiter sont les mots mémoire du code du programme d'application. Supposons que la taille de ce programme d'application soit n mots mémoire.

- La phase 1 de l'algorithme de génération du programme watchdog consiste en un parcours complet des n mots. Le parcours se faisant sans cycle interne, la complexité de la phase 1 est O(n).
- Le tri de la liste des noeuds (destination et séquencement) est d'une complexité O(N log N), où N représente le nombre de noeuds.
- La phase 2 consiste en un parcours complet des n mots et en une recherche d'un élément dans une liste de N éléments. Le parcours est en O(n) et la recherche en O(N log N), où N représente le nombre de noeuds.

Dans le pire des cas, on peut avoir N=n, aussi la complexité générale de l'algorithme est O(n log n).

2.3.3. Optimisations du programme de vérification

Avec la méthode WDP, telle quelle vient d'être décrite, une instruction watchdog est affectée à chaque instruction de séquencement et chaque destination. Or une instruction située immédiatement après un branchement inconditionnel est forcément une destination. Dans ce cas, il existe dans le programme de référence deux noeuds dont les adresses de localisation sont consécutives.

Or, pour un noeud destination qui n'est pas un point de jonction, la seule information réellement nécessaire est l'adresse de localisation du noeud. La valeur de référence associée à ce noeud est totalement arbitraire.

Ainsi, dans le cas d'un branchement inconditionnel, toujours suivi d'une destination, les informations du noeud destination ne sont pas nécessaires au fonctionnement du watchdog :

- l'adresse de localisation de la destination est égale à l'adresse du branchement inconditionnel plus 1,
- la valeur de référence du noeud destination (arbitraire) peut être égale à la valeur de référence du noeud correspondant au branchement conditionnel.

Il est donc possible de supprimer un noeud destination situé immédiatement après un noeud correspondant à un branchement inconditionnel. Pour que le watchdog puisse identifier ce cas, le type du noeud du branchement inconditionnel est modifié (Figure 2.6).

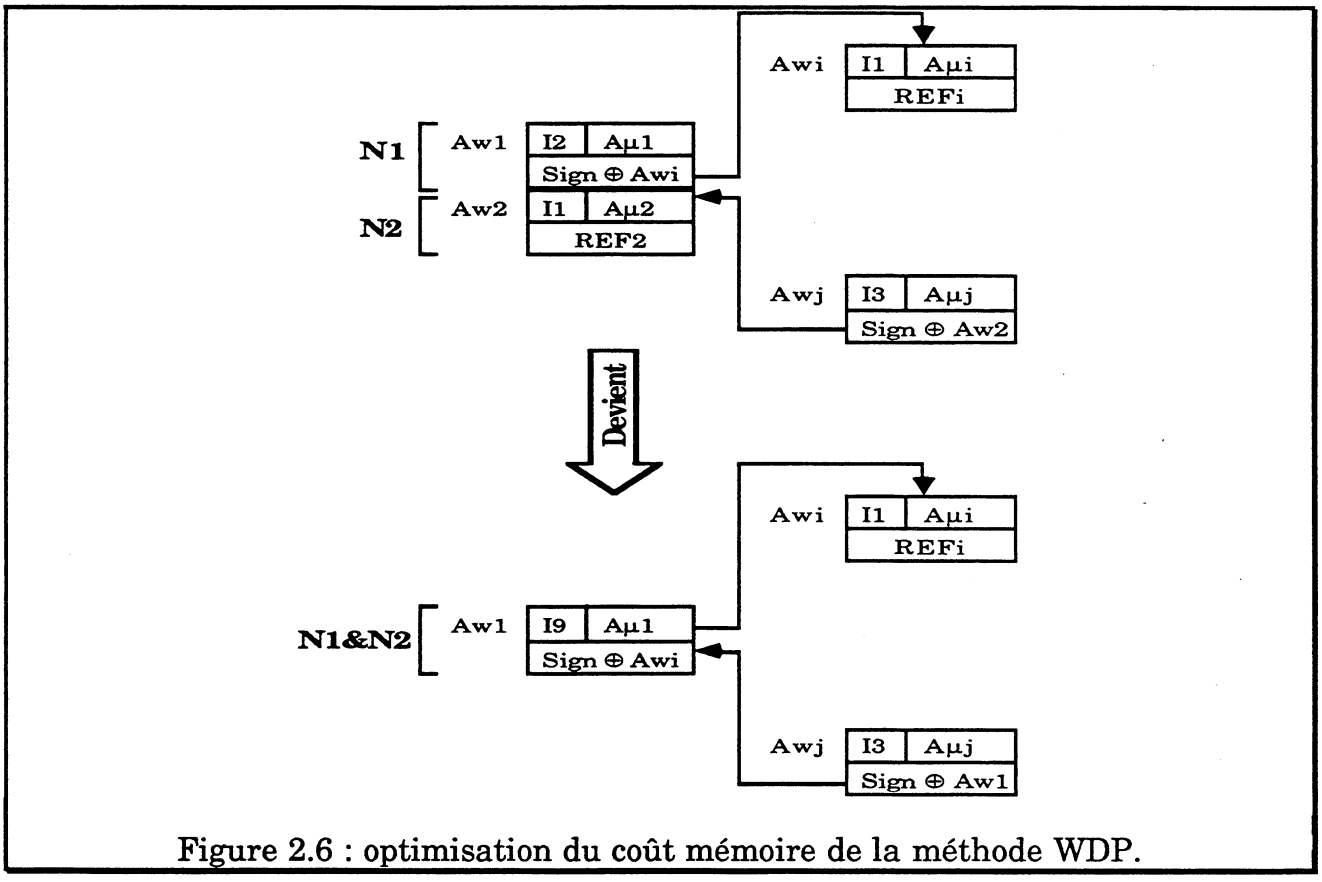


Figure 2.6 : optimisation du coût mémoire de la méthode WDP.

Le traitement associé aux instructions watchdog de type "Branchement inconditionnel suivi de noeud destination" (type I9), est identique à celui des branchements inconditionnels.

Le traitement associé aux branchements dont la destination est une instruction de type I9, par contre, est légèrement modifié. Lorsque le PCW de la destination d'un branchement pointe sur une instruction de type I9, l'adresse microprocesseur à attendre est égale à :

(adresse de localisation du noeud pointé par $PCW_{dest.}$) + 1

et non pas directement l'adresse de localisation du noeud pointé par $PCW_{dest.}$.

Une telle optimisation est aussi applicable aux noeuds de retour inconditionnel de sous-programme qui sont toujours suivis par un noeud destination.

Il faut remarquer que cette optimisation mémoire ne change strictement rien au niveau de la latence et du taux de détection de WDP. Seul le coût mémoire du programme de vérification est diminué, le fonctionnement du watchdog étant quasiment inchangé.

2.4. Capacité de détection d'erreur

La détection des erreurs se fait de différentes manières, suivant qu'il s'agit d'une erreur de bit ou d'une erreur de séquencement, comme dans toutes les méthodes (ESM ou DSM) qui disposent d'un repérage externe des extrémités de blocs (PSAb en est l'exemple le plus typique).

Dans les sections relatives au calcul du taux de couverture, les notations suivantes seront utilisées :

- μ : masquage du dispositif de compaction (formule {P} cf 1.2.2.2.2)
- α : largeur du bus d'adresse microprocesseur,
- ρ : largeur des adresses réduites stockées dans le programme WDP, ($\rho \leq \alpha$),
- k : largeur du bus d'adresse du watchdog = nombre de bits de la signature,
- β : taille moyenne d'un bloc linéaire (en nombre de mots mémoire),
- P : nombre d'instructions de séquencement,
- Q : nombre de destinations.

Le terme "masquage" sera utilisé au sens "non détection" d'une erreur, que ceci soit lié à la méthode de test ou à la compaction d'information.

2.4.1. Erreurs de bits

Les erreurs de bit modifient la signature et sont donc détectées lors de la vérification de la signature au niveau d'un noeud.

Toutefois, dans WDP, la vérification de la signature peut se faire soit directement, soit indirectement suivant le type du noeud qui suit le(s) mot(s) ayant engendré l'erreur :

a/ Si le noeud sur lequel a lieu la vérification est un noeud destination (ou retour d'exception ou de sous-programme), la signature erronée est comparée explicitement à la référence stockée dans le champ 3 de l'instruction WDP (vérification directe). Dans ce cas, le taux de détection (τ_1) ne dépend que de la probabilité de masquage du dispositif de compaction et la latence moyenne est fonction de la taille moyenne d'un bloc linéaire. On a donc :

$$\tau_1 = 1 - \mu \text{ , à latence moyenne } L = \beta$$

b/ Si le noeud sur lequel a lieu la vérification est un noeud de séquencement dont le champ 3 est haché avec l'adresse WDP du noeud destination, l'utilisation de la signature erronée pour le hachage inverse fausse l'adresse WDP du noeud destination.

- Si le branchement est effectué par le microprocesseur, l'adresse microprocesseur de saut est vérifiée en la comparant à celle stockée au niveau de l'instruction pointée par l'adresse WDP calculée (champ 2). L'adresse microprocesseur réelle de saut et l'adresse microprocesseur stockée dans cette instruction WDP ne correspondant pas, l'erreur est détectée (vérification indirecte).
- Si le branchement n'est pas effectué, la vérification de la signature est reportée au noeud suivant.

Le taux de détection d'une erreur de bit sur un noeud de séquencement (branchement effectué) est appelé τ_2. Outre le masquage du dispositif de compaction, un masquage peut se produire à cause de la localisation des noeuds par adresses réduites dans le cas où les bits de poids faibles de l'adresse de localisation (erronée) correspondent à l'adresse de branchement (juste) du processeur. On a donc :

$$\tau_2 = 1 - \left(\mu + \frac{2^{\alpha-\rho} - 1}{2^{\alpha} - 1} \right) \text{ , à latence } L = \beta,$$

Cette probabilité de détection augmente avec la latence. En effet, suite à une telle erreur, le watchdog est décalé dans son programme de vérification par rapport au processeur dont le séquencement n'a pas été modifié. A chaque occurrence d'un noeud, en supposant que la taille des blocs soit identique pour le processeur et dans le programme de vérification (pire cas), la probabilité de détection au niveau d'un noeud est égale à τ_1 ou τ_2 suivant le type du noeud. Dans le pire des cas (τ_2 à chaque noeud), on a donc :

$$\tau_2(L) = 1 - \left(\mu + \frac{2^{\alpha-\rho} - 1}{2^{\alpha} - 1} \right) \left(\chi + \frac{2^{\alpha-\rho} - 1}{2^{\alpha} - 1} \right)^{\left(\frac{L}{\beta} - 1 \right)}$$

Ce taux de détection tend rapidement vers 1 quand L/β augmente.

Il faut remarquer ici que le masquage du dispositif de compaction (μ) n'intervient plus dans le calcul de la probabilité de masquage quand L/β > 1 (deuxième parenthèse). Une valeur (χ), représentative de la corrélation des signatures intermédiaires est utilisée ($\chi \approx 2^{-k}$). Si la signature était réinitialisée avec une valeur fixe à chaque branchement, cette valeur serait différente du fait de la plus forte corrélation des signatures intermédiaires ($\chi > 2^{-k}$).

2.4.2. Erreurs de séquencement

Les erreurs de séquencement sont détectées dans WDP grâce au repérage des extrémités de blocs et non à travers la signature, car l'invariance de la signature est assurée par réinitialisation et non par ajustements. Il faut distinguer deux cas :

a/ Une rupture de séquence du microprocesseur intervient dans un bloc linéaire (branchement inopiné). Le taux de détection (τ_3) pour ce type d'erreur est égal à :

$$\tau_3 = 1 \text{ , à latence } L = 0$$

b/ Un branchement erroné a lieu à un noeud de séquencement. Le PCW obtenu après hachage inverse (avec une signature juste) pointe un noeud dont l'adresse de localisation ne correspond pas à l'adresse de saut réelle du branchement erroné. Le taux de détection de ce type d'erreur est appelé τ_4 . Comme dans le cas d'une erreur de bit sur un noeud de séquencement, un masquage peut se produire à cause de la localisation des noeuds par adresses réduites dans le cas où les bits de poids faibles de l'adresse de localisation (juste) correspondent à l'adresse de branchement (erronée) du processeur. Par contre, le masquage du dispositif de compaction n'intervient pas dans ce cas. On a donc :

$$\tau_4(L) = 1 - \left(\frac{2^{\alpha-\rho} - 1}{2^{\alpha} - 1}\right)\left(\chi + \frac{2^{\alpha-\rho} - 1}{2^{\alpha} - 1}\right)^{\frac{L}{\beta}}$$

Ce taux de détection tend rapidement vers 1 quand L/β augmente.

Tout comme dans le cas des erreurs de bits sur un noeud de séquencement (τ_2), le taux de détection τ_4 augmente avec la latence car le processeur et le watchdog sont décalés. Après le premier masquage, le comportement du système est identique, que l'erreur ait eu pour origine une erreur de bit ou une erreur de séquencement.

2.4.3. Erreur de bits et de séquencement cumulées

Le cas où erreur de bit (signature erronée) et erreur de séquencement (branchement erroné) se combinent doit également être considéré à cause du hachage des signatures de référence.

Un masquage peut se produire dans le cas où le hachage inverse avec la signature erronée donne un PCW qui pointe le noeud destination atteint par le branchement erroné. Contrairement aux autres cas de masquage, ce cas est irrémédiable et le taux de détection n'augmente donc pas avec la latence. La probabilité d'un tel masquage est égale à :

$$M1 = \frac{(Q - 1)}{(2^{\alpha} - 1)} \cdot \frac{1}{(2^{k} - 1)}$$

Dans le pire des cas, on a :

$P + Q = 2^k - 1$ ceci correspond à un espace d'adressage watchdog rempli par le programme de vérification,

et $Q = P$ ceci correspond à un programme comportant autant de destinations que d'instructions de séquencement.

La probabilité de masquage M1 est alors égale à :

$$M1 = \frac{\frac{2^k - 1}{2} - 1}{(2^{\alpha} - 1)(2^{k} - 1)} = \frac{2^k - 3}{2(2^{\alpha} - 1)(2^{k} - 1)} \approx \frac{1}{2^{\alpha+1}}$$

Ce type d'erreur peut également se produire suite à une seule erreur de bit si le seul mot mémoire erroné fausse l'adresse de la destination d'une instruction de séquencement. La probabilité de ce masquage est égale à :

$$M2 = \frac{1}{(2^{\alpha} - 1)} \cdot \frac{1}{(2^{k} - 1)} \approx \frac{1}{2^{\alpha+k}}$$

Un masquage peut également se produire à cause du repérage par adresses réduites dans le cas où seuls les bits de poids faibles de l'adresse de localisation (erronée) correspondent à l'adresse de branchement (erronée) du processeur. Dans ce cas, le taux de détection est égal à τ_4 mais à latence $L = \beta$ pour l'erreur de bit. Le masquage du dispositif de compaction n'intervient pas à latence minimale car, si l'erreur de bit est masquée par la compaction, on retrouve le cas d'une erreur de séquencement seule (taux de détection τ_4 à latence $L = 0$).

2.4.4. Exemple numérique

L'exemple pris ici correspond au cas de l'implantation qui sera décrite en section 2.7 .

- nombre de bits des adresses microprocesseur : $\alpha = 24$
- nombre de bits des adresses watchdog : $k = 16$
- nombre de bits des adresses réduites : $\rho = 12$

Tableau 2.1 : taux de détection de WDP en fonction de la latence (exemple).

L/β	0	1	2	...	∞
τ_1	0	$1 - 2^{-16}$	x	x	x
τ_2	0	$1 - 2^{-12}$	$1 - 2^{-24}$	...	$1 - 2^{-37}$
τ_3	1	x	x	x	x
τ_4	$1 - 2^{-12}$	$1 - 2^{-24}$	$1 - 2^{-36}$	...	$1 - 2^{-37}$
M1	2^{-25}	2^{-25}	x	x	x
M2	2^{-40}	x	x	x	x

Un "x" dans une colonne du tableau 2.1 signifie que la détection ne peut plus avoir lieu passé le premier masquage de l'erreur.

Les taux τ_2 et τ_4 , quand L/β tend vers l'infini sont limités par la probabilité que le watchdog et le processeur se rejoignent à un même noeud par des chemins différents, sans aucune détection. Cette probabilité est quasiment impossible à calculer et on prendra comme pire cas le produit du masquage à latence minimale et du masquage entraîné par le hachage (M1).

$$\text{C'est-à-dire : } 2^{-12} \cdot M1 = 2^{-12} \cdot 2^{-25} = 2^{-37}$$

2.4.5. WDP comparé à d'autres méthodes

L'avantage de la méthode WDP en terme de détection d'erreur vient du fait que la détection des erreurs de séquencement est effectuée explicitement grâce au repérage des extrémités de blocs par leur adresse.

2.4.5.1. Comparaison avec les méthodes sans ajustements

La détection des erreurs de bits se fait pour ces méthodes avec un taux de couverture qui dépend de la probabilité de masquage entraîné par la compaction :

- dans PSAb ou [Saxe 89] :

$$\tau_1 = \tau_2 = 1 - \mu \text{ , à latence } L = \beta$$

- dans OSLC, la signature est comparée à une liste de N références :

$$\tau_1 = \tau_2 = \frac{1 - \mu}{N}, \text{ à latence } L = \beta$$

$$(32 \leq N \leq 256)$$

- dans SIS, le hachage des références avec les adresses de branchement du processeur entraîne un très fort masquage :

$$\tau_1 = \tau_2 = \frac{Q - 1}{2^{\alpha} - 1}, \forall L \geq \beta$$

Ce masquage est de l'ordre de 0,06 à 0,16 [Wilk 87], suivant la taille du programme, ce qui est énorme comparé au masquage introduit par le hachage dans WDP.

La détection des erreurs de séquencement se fait explicitement, pour les méthodes disposant d'un repérage externe des extrémités de blocs linéaires (PSAb, OSLC) et donc, pour les ruptures de séquences inopinées, on a également :

$$\tau_3 = 1 \text{ , à latence } L = 0$$

Cependant, dans ces méthodes, la détection d'un branchement erroné dont la destination est un début de bloc linéaire est impossible, contrairement à WDP où ceci est parfaitement détecté. Pour ces méthodes on a :

$$\tau_4 = \frac{Q - 1}{2^{\alpha} - 1}, \forall L \geq 0$$

Dans SIS et [Saxe 89], des possibilités de masquage existent, quel que soit le type de l'erreur de séquencement :

- dans SIS, le hachage des références avec les adresses de branchement du processeur entraîne un masquage identique à celui des erreurs de bits :

$$\tau_3 = \tau_4 = \frac{Q - 1}{2^{\alpha} - 1}, \forall L \geq 0$$

- dans [Saxe 89], la corrélation des signatures intermédiaires peut masquer certains branchements erronés ou inopinés (par exemple : erreur mémoire transformant une instruction quelconque en branchement inconditionnel) :

$$\tau_3 = \tau_4 = 1 - \chi, \forall L \geq 0$$

χ étant fonction de la distribution des signatures intermédiaires ($\chi >> 2^{-k}$).

2.4.5.2. Comparaison avec les méthodes ESM avec ajustements

Pour le taux de détection des erreurs de bits dans les méthodes avec ajustements, il faut distinguer les cas suivants :

- la même erreur est compactée plusieurs fois avant un test de la signature (erreur permanente dans une boucle sans test),

- plusieurs erreurs sont compactées plusieurs fois avant un test de la signature (erreurs permanentes dans une boucle sans test),
- plusieurs erreurs sont compactées une seule fois. Dans ce dernier cas seulement on a :

$$\tau_1 = \tau_2 = 1 - \mu \text{ , à latence variable}$$

La probabilité de masquage dans les deux premiers cas ne peut être déterminée avec la même fonction μ car le modèle d'indépendance des erreurs ne peut plus être utilisé, les erreurs étant corrélées. Cette probabilité n'a jamais été calculée d'une manière théorique. Elle a seulement été approchée par simulations [Delo 93].

Dans la méthode PSAg, contrairement à [Wilk 87] et Hsurf, un test avec réinitialisation est placé dans chaque boucle du programme ce qui permet d'assurer à la fois l'invariance des signatures de la boucle et une probabilité de masquage égale à μ pour les erreurs de bits dans une boucle.

Par contre, la réinitialisation après un test entraîne une plus forte corrélation des signatures intermédiaires ce qui nuit à la détection des erreurs de séquencement. Le taux de couverture de la méthode PSAg pour les erreurs de séquencement a été évalué dans [Wilk 87] et [Wilk 90] à :

$$99{,}5\% \leq \tau_3 = \tau_4 \leq 99{,}9\% \text{ , à latence variable}$$

Pour [Wilk 87] et Hsurf, la distribution aléatoire des signatures intermédiaires étant supposée, le taux de détection des erreurs de séquencement est égal à (quelle que soit la fonction de compaction) :

$$\tau_3 = \tau_4 = 1 - 2^{-k} \text{ , à latence variable}$$

Cette égalité n'est cependant vraie que dans le cas où toutes les signatures intermédiaires du programme sont différentes [Wilk 88].

2.4.6. Conclusion sur le pouvoir de détection

D'après l'analyse qui vient d'être faite, on peut constater que l'efficacité de la méthode WDP est particulièrement bonne. En particulier, elle possède tous les avantages des méthodes qui assurent l'invariance de la signature par réinitialisation du dispositif de compaction, tout en supprimant le masquage propre à ces méthodes dans le cas d'un branchement erroné ayant comme destination un début de bloc linéaire.

En ce qui concerne les méthodes avec ajustements, il faut remarquer que le calcul théorique d'un taux de détection se heurte à des problèmes de corrélation difficiles à résoudre. Le taux de détection de $1 - 2^{-k}$ avancé par certains auteurs [Wilk 87] n'est en fait que la limite supérieure du taux de détection d'une méthode

ESM avec ajustements. Personne n'est actuellement en mesure de fournir une garantie sur la limite inférieure de ce taux de détection.

La méthode WDP est donc particulièrement intéressante au point de vue de l'efficacité car elle résout à la fois le problème des méthodes avec ajustements (possibilité de calcul du taux de masquage) et le problème des méthodes sans ajustements (élimination du masquage de certaines erreurs de séquencement).

2.5. Calcul du coût mémoire

Dans la méthode de comparaison qui suit, décrite dans [Wilk 88] et [Wilk 90], on suppose que le code du programme d'application vérifié a été généré avec un compilateur à partir d'une description en langage de haut niveau (HLL : High Level Language). Les structures de contrôle de séquencement habituellement utilisées dans les langages de haut niveau sont : If-Then, If-Else, While-For, Repeat-Until (Do), Switch (à trois cas), Call et Return (cf Annexe 2).

Le tableau 2.2 fournit la fréquence relative d'utilisation des différentes structures classiques des langages de haut niveau [Wilk 87].

Tableau 2.2 : fréquence d'utilisation des structures classiques.

If then	If Else	Switch	While/For	Do	Call	Return
0,23	0,14	0,02	0,13	0,00	0,38	0,10

Dans ce qui suit, nous supposons que le microprocesseur manipule des mots de 16 bits. Les instructions WDP tiennent sur deux mots, les instructions Cerb-16 sur 2 ou 3 mots selon leur type, les singularités repérées par instructions dédiées sur deux mots et les références repérées par étiquette sur un mot.

Les méthodes comparées sont celles décrites en section 1.4.4 (méthodes ESM). La seule méthode DSM présentée ici est Cerberus 16, le calcul du coût mémoire pour OSLC et ASIS ne peut être fait que globalement sur l'ensemble d'un programme et il n'est donc pas possible avec le mode de calcul utilisé ici. D'une manière générale, le coût mémoire de OSLC est toujours supérieur à celui de PSAb (pour une efficacité toujours inférieure) et le coût mémoire de ASIS est comparable à celui de Sax89 [Saxe 89].

Les méthodes comparées sont donc :
- méthode PSAb [Namj 82],
- méthode PSAg [Namj 82],
- méthode SIS [Shen 83],
- méthode Wil87 [Wilk 87] (nombre d'ajustements minimal),

- méthode Arcs [Wilk 87], avec un ajustement sur tous les arcs de séquencement (méthode de base pour un moniteur externe),
- méthode Hsurf [Jay 86] (nombre d'ajustements minimal),
- méthode Sax89 [Saxe 89],
- méthode Cer16 (Cerberus 16) [Namj 83],
- méthode WDP , telle qu'elle est décrite en section 2.2,
- méthode Wopt , c'est-à-dire WDP avec optimisation du programme de vérification.

2.5.1. Coûts mémoire minimums

Le tableau 2.3 présente le nombre de mots mémoires nécessaires aux références pour chaque structure et chaque méthode de test. Les nombres entre parenthèses du tableau 2.3 représentent le nombre de points de test de la signature par bloc linéaire pour chaque méthode et chaque structure. Ce nombre de points de test définit la latence maximale d'une méthode. Le coût présenté dans cette section est donc un coût minimum.

Tableau 2.3 : matrice des coûts mémoire verticaux minimum par structure et par méthode (en mots).

	Méthodes ESM							Méthodes DSM		
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	**WDP**	**Wopt**
if then	2(1)	1(0)	2(1)	1(0)	1(0)	2(0)	6(1)	5(1)	**4(1)**	**4(1)**
if else	3(1)	1(0)	2(1)	1(0)	2(0)	2(0)	9(1)	8(1)	**8(1)**	**6(1)**
switch(3)	6(1)	3(0)	2(0,5)	3(0)	5(0)	6(0)	18(1)	17(1)	**16(0,5)**	**12(0,5)**
while/for	3(1)	2(1)	2(1)	1(0)	2(0)	2(0)	9(1)	8(1)	**8(1)**	**6(1)**
do	2(1)	2(1)	2(1)	1(0)	1(0)	2(0)	6(1)	5(1)	**4(1)**	**4(1)**
call	1(1)	1(1)	0(1)	1(0)	1(0)	2(0)	3(1)	3(1)	**2(1)**	**2(1)**
return	1(1)	1(1)	1(1)	1(1)	1(1)	0(0)	3(1)	2(1)	**2(1)**	**2(1)**

La méthode PSAb sert de référence car elle ajoute une signature de référence par bloc linéaire. La colonne correspondant à PSAb dans le tableau 2.3 décrit donc le nombre de blocs linéaires par structure.

Le tableau 2.4 donne le coût mémoire vertical pour chaque méthode. Ce coût correspond au nombre de mots mémoire ajouté. Il ne tient pas compte de la présence éventuelle d'une mémoire étiquette (coût horizontal) pour les méthodes PSAb et PSAg, mais également pour Wil87 dans le cas d'un moniteur externe.

Le coût mémoire vertical de chaque méthode est calculé de la manière suivante [Wilk 87] :

- Le produit matriciel du tableau 2.2 et de la première colonne du tableau 2.3 fournit la moyenne pondérée (par le poids moyen de chaque structure classique) du nombre de blocs linéaires par structure (mpb).
- En multipliant ce résultat par le nombre moyen de mots par bloc (4 à 10) [Wilk 90], on obtient la moyenne pondérée de mots par structure (mpm).
- En effectuant le produit matriciel du tableau 2.2 par chaque colonne du tableau 2.3, on obtient pour chaque méthode la moyenne pondérée du nombre de mots ajoutés par structure (mpma).
- En comparant ce résultat à mpm, on obtient pour chaque méthode le coût mémoire vertical.

Le coût mémoire vertical est exprimé, dans le tableau 2.4, en pourcentage de mots mémoire nécessaire à une méthode par rapport à la taille du programme vérifié.

Tableau 2.4 : coûts mémoire verticaux minimum par méthodes (en %).

	Méthodes ESM							Méthodes DSM		
méthode	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	**WDP**	**Wopt**
coût (%)	10-25	6-16	6-15	6-14	7-18	10-25	30-75	27-67	**23-58**	**20-50**

La figure 2.7 représente graphiquement les informations du tableau 2.4. On pourra constater que les méthodes ESM proposent un coût mémoire minimum largement inférieur à celui des méthodes DSM à l'exception notable de Sax89. Cette méthode utilise en effet des signatures de référence d'une taille deux fois supérieure aux autres méthodes, avec un repérage par instructions dédiées et une référence par bloc linéaire.

Il faut remarquer que parmi les méthodes offrant le coût mémoire vertical le plus faible certaines nécessitent une mémoire étiquette en parallèle de la mémoire contenant le programme (coût mémoire "horizontal" : PSAb, PSAg, Wil87).

Le tableau 2.5 donne une estimation de la latence moyenne de détection des erreurs pour chaque méthode lorsque le coût mémoire est minimal. Cette latence correspond au nombre moyen de mots que le microprocesseur a le temps de charger avant que le moniteur ou le watchdog ne puisse éventuellement détecter l'erreur.

Tableau 2.5 : latence moyenne de détection pour chaque méthode (en nombre de mots).

	Méthodes ESM							Méthodes DSM		
méthode	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	**WDP**	**Wopt**
latence	4 - 10	7 - 17	7 - 17	39 - 95	39 - 95	∞	4 - 10	4 - 10	**7 - 17**	**7 - 17**

Le calcul de la latence d'une méthode est effectué d'une manière similaire au calcul du coût mémoire mais en utilisant les informations relatives au nombre de points de test par structure et non les informations relatives au nombre de mots mémoire ajouté.

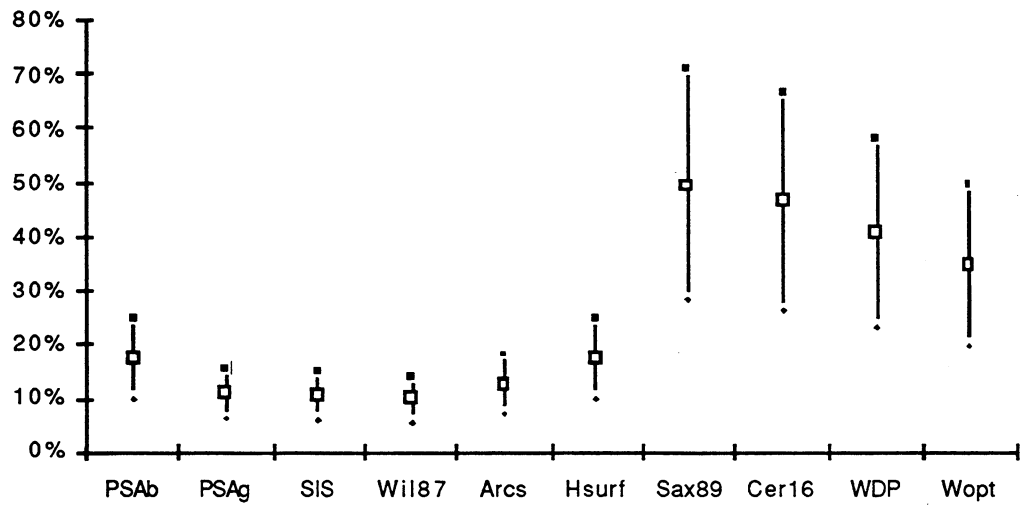


Figure 2.7 : coût mémoire vertical minimum de WDP comparé aux autres méthodes.

On pourra remarquer, sur les tableaux 2.4 et 2.5, que la méthode WDP optimisée présente un coût mémoire rigoureusement égal au double de PSAb (une instruction, soit deux mots mémoire, par bloc linéaire pour WDP) et une latence égale à celle de SIS (même phénomène d'augmentation de la latence par rapport à PSAb, entraîné par le hachage de certaines références qui supprime un test en cas de branchement conditionnel non effectué).

Pour la plupart des méthodes considérées, la latence peut être réduite, par ajout de points de test mais au prix d'une augmentation du coût mémoire. Dans la section suivante, des coûts mémoire à latence identique sont donc présentés.

2.5.2. Coûts mémoire à latence égale

Le tableau 2.6 présente les coûts mémoire verticaux des différentes méthodes pour une latence de détection égale à un test par bloc linéaire, c'est-à-dire la latence de la méthode PSAb qui est donc encore prise comme référence. Pour les autres méthodes, l'introduction de points de test supplémentaires se traduit par l'ajout de :

- 2 mots mémoire, pour les méthodes SIS, Hsurf, Wil87 et Arcs : un mot de repérage plus un mot pour la signature de référence,
- 1 mot mémoire, pour PSAg : une signature de référence repérée par étiquette,
- 0 mot mémoire, pour Cer16 et Sax89 qui, comme PSAb testent la signature à chaque bloc linéaire,
- 2 mots mémoire, pour WDP : noeud destination fictif.

Les tableaux 2.6 et 2.7 sont les équivalents des tableaux 2.3 et 2.4 mais à latence égale. Ils représentent respectivement le nombre de mots mémoire par structure et le coût mémoire vertical pour chaque méthode.

Tableau 2.6 : matrice des coûts mémoire verticaux à latence égale par structure et par méthode (en mots).

	Méthodes ESM							Méthodes DSM		
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	WDP	Wopt
if then	2	2	2	3	3	4	6	5	**4**	**4**
if else	3	3	2	5	6	6	9	8	**8**	**6**
switch(3)	6	6	4	9	11	12	18	17	**22**	**18**
while/for	3	2	2	5	6	6	9	8	**8**	**6**
do	2	2	2	3	3	6	6	5	**4**	**4**
call	1	1	0	3	3	4	3	3	**2**	**2**
return	1	1	1	1	1	2	3	2	**2**	**2**

Tableau 2.7 : coûts mémoire verticaux à latence égale par méthodes (en %).

	Méthodes ESM							Méthodes DSM		
méthode	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	**WDP**	**Wopt**
coût (%)	10-25	9-23	6-16	19-46	20-50	24-60	30-75	27-67	**24-60**	**21-52**

La figure 2.8 est une représentation graphique du tableau 2.7 . On pourra constater que le coût mémoire des méthodes avec ajustements augmente sensiblement alors que les autres méthodes sont, en proportion, moins affectées. Ceci confirme que les méthodes qui réinitialisent la signature pour assurer son

invariance proposent un bon compromis entre le coût mémoire et la latence de détection (excepté Sax89 qui est un cas très particulier).

On pourra remarquer, sur le tableau 2.7 et sur la figure 2.8, que le coût mémoire de WDP reste toujours inférieur à celui de Sax89 et Cer16. Par ailleurs, le coût mémoire de WDP, dans sa version optimisée, est du même ordre de grandeur que le coût à latence égale de certaines méthodes ESM avec ajustement (Arcs et Hsurf).

L'analyse de la figure 2.8 permet également de faire quelques remarques sur le choix d'une méthode en fonction de critères autres que le simple coût mémoire vertical.

Parmi les méthodes dont le coût mémoire vertical est le plus faible (PSAb, PSAg, SIS), seule SIS ne nécessite pas de mémoire étiquette, ce qui en fait donc une méthode particulièrement économique à tout point de vue. Son principal inconvénient est bien sûr son taux de détection qui est franchement inférieur à celui de toutes les autres méthodes (cf 2.4.5).

A l'inverse, le coût mémoire vertical particulièrement élevé de la méthode Sax89 ne doit pas occulter le fait qu'il s'agit de la seule méthode ESM ayant une efficacité comparable à celle de PSAb (pas besoin de dispositif externe pour détecter les cas où un test n'a jamais lieu) mais qui, à la différence de PSAb, ne nécessite pas de mémoire étiquette pour le repérage des extrémités de blocs linéaires.

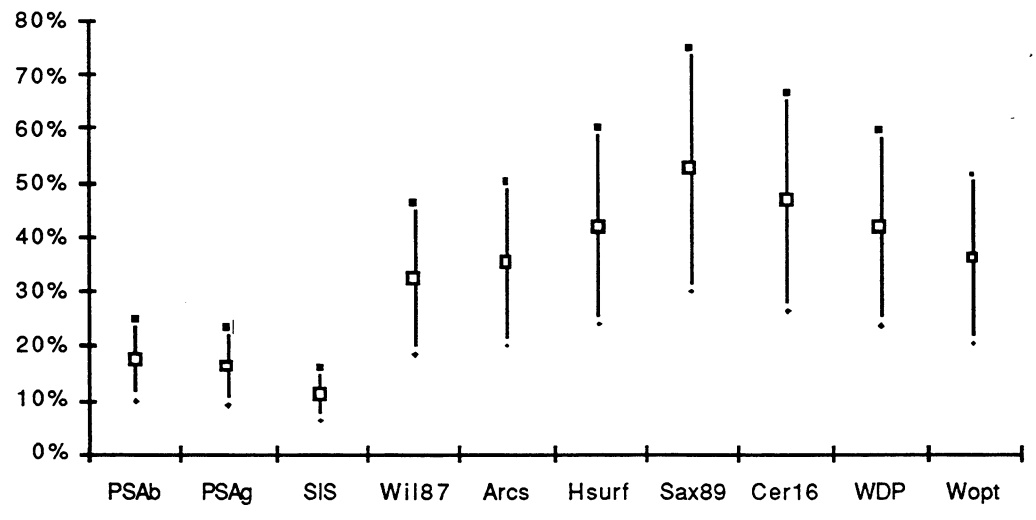


Figure 2.8 : coût mémoire vertical de WDP, à latence égale, comparé aux autres méthodes.

2.5.3. Conclusion sur le coût mémoire de WDP

On constate d'après les données précédentes, que le coût mémoire minimal des méthodes utilisant un programme watchdog séparé (Cer16, WDP) est plus important que celui de la plupart des méthodes qui insèrent les signatures de référence et les ajustements dans le code du programme vérifié. Ceci vient du fait que le programme watchdog doit stocker, en plus des informations relatives au test de la signature, des informations sur la structure du programme vérifié.

La méthode WDP, et à plus forte raison Wopt, présentent toutefois des coûts mémoire inférieurs à Cer16, seule méthode DSM comparable, mais également inférieurs à Sax89 qui est de type ESM.

Toutefois, il faut noter l'influence que peuvent avoir les valeurs données dans le tableau 2.2 sur le coût mémoire des différentes méthodes. Ainsi, la méthode WDP sera désavantagée pour les programmes ayant une proportion importante de structures "Switch". Par contre elle est nettement avantagée pour les programmes comportant une proportion plus importante de structures "Do", "Return" ou "If Then".

Par ailleurs, la comparaison des coûts mémoire à latence égale a montré que la méthode WDP supporte tout à fait la comparaison avec certaines méthodes ESM. Le compromis entre le coût mémoire et le pouvoir de détection est donc assez favorable à la méthode WDP.

2.6. Prise en compte des caractéristiques du processeur vérifié

Les microprocesseurs actuels, 16 ou 32 bits, disposent souvent de mécanismes gênants pour les méthodes de vérification du flot de contrôle par analyse de signature dérivée, dans le cas d'un dispositif externe au processeur vérifié.

Pour toutes ces méthodes, la présence d'une ante-mémoire (mémoire cache) pour les instructions rend impossible la génération d'une signature dérivée par un dispositif externe. Ce cas ne sera donc pas détaillé plus que cela.

Dans le cas de la méthode WDP[1], du fait du repérage des singularités par leur adresse, il faut considérer le cas où le processeur vérifié dispose d'une MMU (Memory Management Unit) intégrée. Ce cas sera détaillé en section 2.6.1 .

Pour toutes les méthodes, la présence d'une lecture anticipée des instructions (processeur pipeline), complique la génération de la signature car alors les

---

1 Ceci est également valable pour la méthode OSLC.

instructions lues ne sont pas forcément exécutées par le processeur. Ce cas sera détaillé en section 2.6.2 . L'adaptation de la méthode WDP à un processeur pipeline sera décrite en section 2.6.2.4 .

Parmi les caractéristiques d'un processeur gênantes pour l'analyse de signature, on peut également citer les départ en exceptions du microprocesseur (interruptions matérielles ou logicielles, "trap"). Ce cas sera détaillé en section 2.6.3 .

2.6.1. Processeur avec MMU intégrée

La grande majorité des programmes ne sont pas écrits en adresses physiques mais en adresses logiques. C'est-à-dire que la position du code dans le plan mémoire n'est pas fixe mais est déterminé par un mécanisme spécial (MMU) qui convertit les adresses logiques émises par le microprocesseur en adresses physiques destinées à la mémoire. Un mécanisme de pagination est souvent associé à une MMU pour travailler en mémoire virtuelle. La figure 2.9 présente un exemple de translation d'adresses par MMU avec mémoire paginée (Source : Intel 80386).

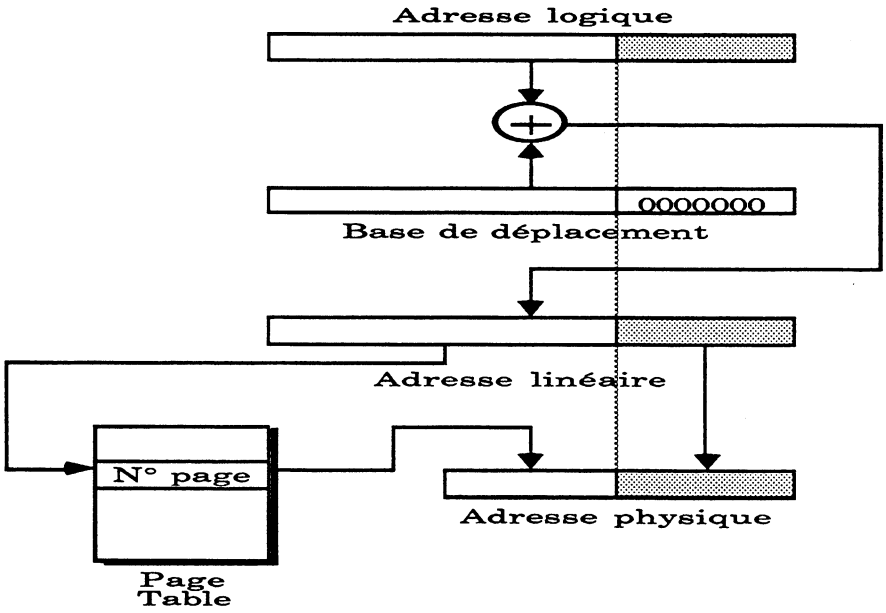


Figure 2.9 : exemple de conversion d'une adresse logique
en adresse physique avec pagination de la mémoire (Intel 80386).

La méthode WDP repose sur la vérification des adresses du programme et celles-ci doivent donc être connues avant la génération du programme watchdog et ne doivent pas varier pendant l'exécution du programme. Le processeur watchdog doit donc avoir accès aux adresses logiques du microprocesseur et non aux adresses

physiques sortant d'une MMU. Ceci pose un problème très gênant dans le cas où la MMU est intégrée au microprocesseur (80286 et 80386 par exemple).

Dans le cas d'une MMU intégrée au processeur, une méthode possible est de laisser une partie de l'adresse invariante (poids faible) par la transformation de l'adresse logique en adresse physique. Seule cette partie invariante sera stockée dans le programme watchdog pour la localisation des noeuds avec une adresse réduite. Le nombre de bits laissés invariants est arbitraire mais il ne peut en aucun cas être supérieur au nombre de bits définissant la taille d'une page dans le cas d'une mémoire paginée.

Par exemple, dans le cas du 80386 d'Intel, la taille du champ de repérage pourra être de 12 bits au plus étant donné que la taille d'une page est 4 Koctets (2^{12} bits). Par ailleurs, les bases de déplacement d'adresses de la MMU devront avoir ces bits de poids faible à zéro pour que ces bits ne soient pas modifiés lors de la transformation des adresses logiques en adresses physiques (Figure 2.9). Il s'agit là d'une contrainte au niveau du système d'exploitation qui ne remet pas en cause la compatibilité du code. Cette contrainte est d'ailleurs satisfaite d'elle-même dans la plupart des cas étant donné que les segments mémoire code sont généralement alignés sur une frontière de page.

Un problème se pose toutefois dans le cas d'une mémoire paginée. En effet, lors d'un changement de page dans les adresses du programme en cours d'exécution, il se peut que les adresses physiques des deux pages ne soient pas consécutives. Pour éviter que le watchdog WDP signale ce cas de figure comme une rupture de séquence inopinée, il faut que celui-ci ne tienne pas compte des bits de poids forts de l'adresse du processeur dans le cas d'un passage à zéro de tous les bits de poids faible de l'adresse lors de l'incrémentation de l'adresse du processeur.

En cas d'erreur de pagination, le masquage introduit est égal à la probabilité d'une erreur de bit et d'une erreur de séquencement cumulées (M1 cf 2.4.3).

2.6.2. Processeurs avec lecture anticipée des instructions

Pour la plupart, les processeurs disposent d'une lecture anticipée des instructions (processeurs "pipeline"), avec une file d'attente dans laquelle les instructions sont placées entre le moment de leur lecture et le moment de leur décodage. Certains processeurs disposent en plus d'une file d'instructions décodées.

Pour un générateur de signature externe au processeur, la compaction des instructions ne peut se faire qu'au moment de leur lecture. Cependant, il existe des cas où une instruction placée dans la file d'attente du processeur n'est pas exécutée, par exemple dans le cas d'une rupture de séquence (bulles dans le pipeline). Ces

instructions lues sont pourtant compactées par le dispositif de génération de signature, ce qui peut poser des problèmes de cohérence pour la signature.

Avant d'aller plus dans le détail des solutions permettant de remédier à ces problèmes, quelques définitions nécessaires sont présentées en section 2.6.2.1 .

2.6.2.1. Définitions

Définition 2.3 : une "unité de programme" est un mot mémoire du code exécutable d'un programme. Suivant le format des instructions d'un processeur, une unité de programme peut contenir plusieurs instructions, ou une instruction peut tenir sur plusieurs unités de programme. Pour certains processeurs (RISC en particulier) l'unité de programme est une instruction.

Définition 2.4 : on appellera "fetch" l'action du processeur qui consiste à charger une unité de programme.

Définition 2.5 : le nombre α est le nombre minimum de fetch qui sépare la fin du chargement d'une instruction et le début de son exécution.
α=nombre minimum d'unités programme que la file d'attente du processeur doit contenir.

Définition 2.6 : le nombre β est le nombre maximum d'unités programme que la file d'attente du processeur peut contenir.

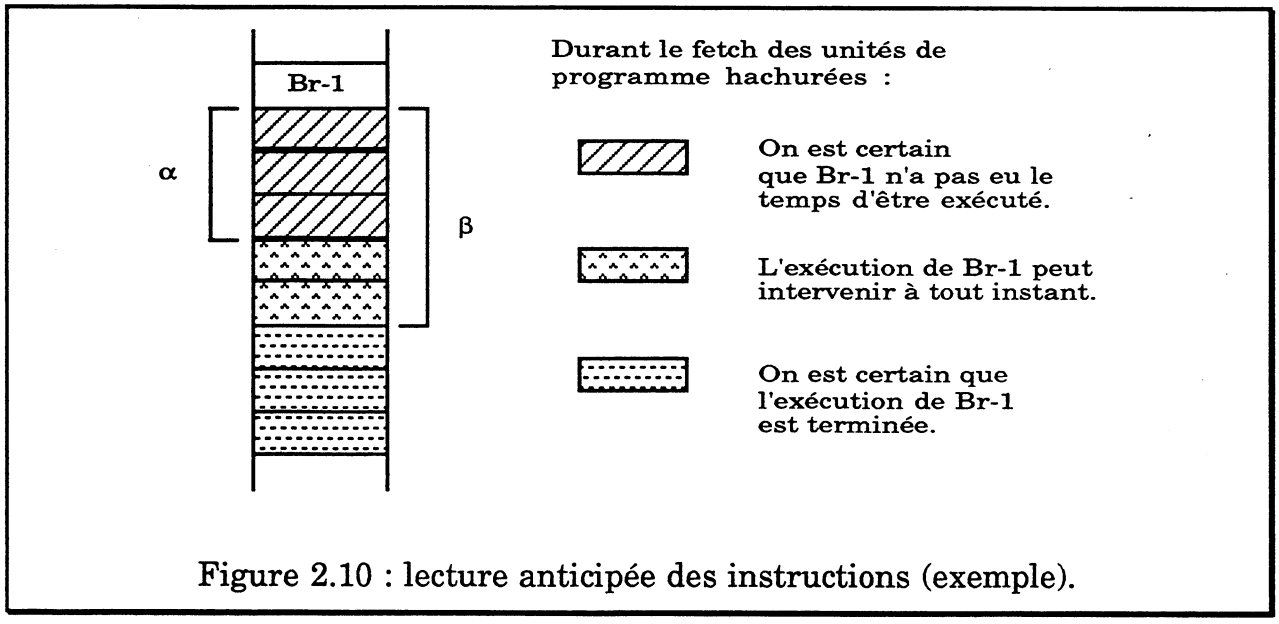


Figure 2.10 : lecture anticipée des instructions (exemple).

Définition 2.7 : le nombre γ est le nombre maximum d'instructions de séquencement que peut contenir la file d'attente du processeur. Ce

nombre n'est différent de β que si l'unité de programme n'est pas l'instruction.

Définition 2.8 : l'"age d'une instruction" est le nombre de fetch ayant eu lieu entre le début du chargement de cette instruction et l'exécution de celle-ci.

Dans la suite de cette section, on distinguera deux cas de processeurs pipelines :

- processeurs avec un pipeline synchrone : $\alpha = \beta - 1$,
- processeurs avec un pipeline asynchrone : $\alpha \leq \beta - 1$.

Dans le premier cas, l'instant de l'exécution est prévisible par un dispositif externe. En effet, une instruction exécutée l'est toujours lorsqu'elle atteint l'age α .

Dans le second cas, au contraire, une instruction exécutée peut avoir un age compris entre α et β .

Les problèmes que pose la présence d'une file d'attente d'instructions, quels que soient le type de pipeline du processeur et la méthode de vérification utilisée, sont de deux types :

- Problème de cohérence de la signature en cas de rupture de séquence, à cause des instructions lues mais non exécutées. Ce cas sera détaillé en section 2.6.2.2 .
- Possibilité d'avoir plusieurs singularités dans la file d'attente du processeur. Ce cas sera détaillé en section 2.6.2.3 .

2.6.2.2. Cohérence de la signature en cas de branchement

La cohérence de la signature nécessite de différencier les cas de pipelines synchrones et de pipelines asynchrones, ainsi que la méthode utilisée pour l'invariance de la signature.

a/ Pipelines synchrones

Dans ce cas, le nombre d'unités de programme chargées après une instruction de séquencement est toujours identique. Ces mots mémoire peuvent donc facilement être pris en compte lors du calcul des références, et ce quelle que soit la manière d'assurer l'invariance de la signature. Une justification formelle, dans le cas d'une méthode avec ajustements sur noeuds peut être trouvée dans [Delo 91].

b/ Pipelines asynchrones

Dans ce cas, le nombre d'unités de programme chargées après une instruction de séquencement n'est pas toujours identique. Il dépend des conditions de fonctionnement du système avant l'exécution d'une instruction de séquencement

(remplissage de la file d'instructions en fonction de la disponibilité du bus mémoire par exemple). Il n'est donc pas possible de prendre en compte les mots mémoires situés après une instruction de séquencement dans le calcul des références.

Une méthode avec ajustements sur noeuds n'est ainsi pas envisageable pour un processeur ayant un pipeline asynchrone. De même, toutes les méthodes dans lesquelles il existe des instructions de branchement qui ne sont pas des singularités ne fonctionnent pas avec ce type de pipeline (Sax 89, Cerberus 16, ASIS).

Une méthode avec ajustements sur arcs, par contre, peut fonctionner, mais uniquement dans les cas où il n'existe pas d'ambiguïté sur la singularité ayant entraîné la rupture de séquence, c'est-à-dire si une seule instruction de séquencement est présente dans la file d'attente du processeur (cf 2.6.2.3).

Les seules méthodes qui ne sont pas trop affectées par la présence d'un pipeline asynchrone sont les méthodes qui réinitialisent la signature en cas de branchement. En effet, avec ces méthodes, la compaction des mots mémoire après la vérification de la signature en fin de bloc (au niveau de l'instruction de séquencement) sera occultée par la réinitialisation du dispositif de compaction au moment de la rupture de séquence (si elle a lieu). Cependant, pour les méthodes qui détectent les ruptures de séquence inopinées, il est nécessaire de modifier leur fonctionnement, car un branchement doit pouvoir se produire entre α et β mots après une instruction de séquencement et non pas au niveau d'une instruction particulière comme dans les cas de pipeline synchrone ou de l'absence de pipeline. Ceci peut évidemment entraîner des cas de masquage supplémentaires. WDP fait partie de ces méthodes.

Il faut remarquer que la seule méthode dont le fonctionnement et le taux de détection ne sont pas affectés par la présence d'un pipeline asynchrone est la méthode SIS (mais le taux de détection est déjà très mauvais), car le test de la signature est toujours réalisé au moment de l'exécution du branchement du processeur. En effet, la signature est en fait testée indirectement à travers l'adresse de branchement du processeur, donc au moment de la rupture de séquence, et ceci sans modification du fonctionnement du moniteur.

2.6.2.3. Présence de plusieurs singularités dans la file d'attente du processeur

a/ Pipelines synchrones

La présence de plusieurs singularités dans la file d'attente du processeur ne pose pas de problèmes particuliers, pour toutes les méthodes, dans le cas d'un pipeline synchrone. En général, le traitement d'une singularité peut se faire

indifféremment au moment de la lecture ou au moment de l'exécution d'une singularité.

b/ Pipelines asynchrones

Dans ce cas, les méthodes avec ajustements sur arcs posent un problème. En effet, il faut, lors d'une rupture de séquence, être en mesure d'identifier l'instruction de branchement à l'origine de la rupture de séquence pour déterminer quelles doivent être la valeur d'ajustement et la signature intermédiaire à utiliser pour le calcul de la signature à la destination du processeur. Ceci est rigoureusement impossible dans le cas où plusieurs instructions de séquencement peuvent se trouver à des adresses distantes de moins de $\beta - \alpha$. Ceci dépend du processeur et de son format d'instructions. Dans tous les cas, il est possible d'ajouter des NOPs entre deux instructions de séquencement pour remédier à ce problème.

Pour les méthodes qui réinitialisent la signature en cas de branchement, la présence de plusieurs singularités dans la file du processeur ne pose aucun problème si la valeur de réinitialisation est fixe (PSAb, SIS, OSLC).

WDP ne fait pas partie de ces méthodes car la signature est réinitialisée avec la référence du noeud destination. Par ailleurs, la gestion du pointeur du watchdog nécessite que tout branchement donnant lieu à une rupture de séquence soit parfaitement identifié, comme dans le cas des ajustements sur arcs, mais ici il n'est pas possible de modifier le programme pour résoudre les cas critiques. Par contre, dans WDP, l'origine d'une rupture de séquence peut être identifiée grâce à l'adresse de sa destination.

La méthode WDP s'applique donc quel que soit le type de pipeline du processeur mais elle nécessite quelques adaptations qui vont être décrites dans la section suivante.

2.6.2.4. Adaptation de WDP aux processeurs pipelines

Les adaptations de WDP décrites dans cette section correspondent à la prise en compte d'un pipeline (synchrone ou asynchrone), d'une manière à entraîner un surcoût matériel aussi faible que possible. Cette solution est par ailleurs celle qui a été implantée dans le cas du watchdog WDP pour un MC68000 dont la description sera faite en section 2.7.3 .

2.6.2.4.1. Fonctionnement

La mise en œuvre de cette méthode impose que le processeur dispose d'un moyen permettant de suspendre momentanément son exécution, ceci pour les cas où le watchdog n'a pas le temps de traiter son programme de vérification assez

rapidement. Un arrêt momentané du processeur peut être imposé en gérant, par exemple, le signal de validation des échanges mémoire du processeur.

La solution décrite ici nécessite également que le watchdog conserve toutes les signatures intermédiaires des instructions chargées par le processeur mais non encore exécutées. Ceci impose une liste de signatures de taille β . On verra en section 2.6.3 que cette liste est également nécessaire pour le traitement des exceptions.

Le principe de la solution décrite ici consiste à ne traiter qu'un noeud à la fois, même dans le cas où plusieurs singularités se trouvent dans la file d'attente du processeur.

Pour les noeuds destination, le fonctionnement du watchdog n'est pas modifié, ceux-ci étant toujours traités au moment de leur chargement.

Pour les noeuds séquencement, le traitement s'étale sur tout le temps pendant lequel l'instruction de séquencement est présente dans le pipeline du processeur, c'est-à-dire pendant au plus β fetchs consécutifs après le chargement de cette instruction.

Pour les noeuds correspondant à des instructions de séquencement inconditionnelles, le watchdog dispose de α fetchs pour charger le noeud destination de cette instruction de séquencement.

- Si une rupture de séquence intervient pendant les fetchs antérieurs à α , il s'agit d'un branchement inopiné.
- Si une rupture de séquence intervient pendant les fetchs compris entre α et β après le chargement de l'instruction de séquencement, le watchdog vérifie que la destination du branchement correspond au champ référence du noeud destination qu'il vient de charger. Dans le cas contraire, il s'agit d'un branchement inopiné.
- Si une rupture de séquence n'est pas intervenue β fetchs après le chargement de l'instruction de séquencement, le watchdog signale une erreur car le branchement aurait du avoir lieu.

Pour les noeuds de séquencement conditionnels, le watchdog dispose de α fetchs pour charger le noeud destination de cette instruction de séquencement. Si une rupture de séquence intervient pendant les fetchs antérieurs à α , il s'agit là aussi d'un branchement inopiné.

Deux cas sont alors à considérer :

Cas N° 1 :

Une rupture de séquence intervient pendant les fetchs compris entre α et β après le chargement de l'instruction de séquencement. Le watchdog vérifie que la destination du branchement correspond au champ référence du noeud destination qu'il vient de charger. Dans le cas contraire, le watchdog considère dans un premier temps que le branchement n'a pas été pris et passe alors au noeud suivant. Cependant, le noeud suivant peut déjà avoir été chargé dans la file d'attente du processeur et, si c'est cas, le watchdog ne peut pas traiter la singularité au moment de son chargement par le processeur. Ceci nécessite un "traitement à retard des singularités" :

Pour déterminer si l'instruction correspondant au prochain noeud est déjà dans le pipeline du processeur, le watchdog effectue une soustraction entre l'adresse du dernier fetch du processeur et l'adresse de localisation du noeud à traiter.

- Si le résultat de la soustraction est négatif, la singularité n'a pas encore été chargée par le processeur et n'a donc pas pu provoquer la rupture de séquence qui est donc un cas de branchement inopiné.
- Si le résultat de la soustraction est inférieur à α , l'instruction n'a pas encore pu être exécutée et il s'agit là encore d'un branchement inopiné.
- Si le résultat de la soustraction est compris entre α et β , l'instruction correspondant au noeud est présente dans le pipeline du processeur et a pu être exécutée. L'age de cette instruction est égal au résultat de la soustraction. La signature intermédiaire au niveau de cette instruction peut être retrouvée dans la liste des signatures à la position correspondant à l'age de l'instruction. Le watchdog traite alors la singularité, et s'il s'agit d'un noeud de séquencement, il reprend le traitement au niveau du cas N°1.

Pendant tous ces traitements, le processeur est arrêté tant que le watchdog n'a pas réussi à identifier l'origine de la rupture de séquence.

Cas N°2 :

Aucune rupture de séquence n'a lieu pendant les fetchs compris entre α et β après le chargement de l'instruction de séquencement. Le watchdog passe alors au noeud suivant et, pour déterminer si l'instruction correspondant à ce noeud est déjà dans le pipeline du processeur, le watchdog effectue là aussi une soustraction entre l'adresse du dernier fetch du processeur et l'adresse de localisation du noeud à traiter.

- Si le résultat de la soustraction est négatif, la singularité n'a pas encore été chargée par le processeur et celui-ci attend donc son chargement avant de pouvoir la traiter.
- Si le résultat de la soustraction est inférieur à β , l'instruction a déjà été chargée par le processeur et le watchdog traite alors la singularité, avec la signature retrouvée grâce au calcul de l'âge de l'instruction. Suivant le type du noeud ainsi traité, le watchdog attend alors une rupture de séquence ou bien il reprend le traitement au niveau du cas N°2.

Il faut donc remarquer que l'adaptation du processeur watchdog à la vérification d'un processeur pipeline est assez minime.

En terme de complexité matérielle, la seule différence concerne l'ajout des dispositifs suivant :

- une file de signatures intermédiaires de longueur β, (cette file est de toute façon indispensable pour le traitement des exceptions et ce dans toutes les méthodes à base de signatures dérivées),
- un mécanisme de calcul de l'age d'un noeud (ceci nécessite de faire la comparaison des adresses avec un soustracteur et non plus avec un comparateur à base de OU Exclusifs).

Au niveau du fonctionnement du processeur, la seule différence consiste à traiter un noeud non pas avec la signature courante, mais avec la signature correspondant à l'age du noeud dans la file des signatures intermédiaires.

La solution décrite ici n'est pas la seule possible. En particulier, si le fonctionnement du watchdog doit être accéléré dans le cas de noeuds consécutifs, plusieurs méthodes existent et ont été décrites dans [Mich 91]. Elles ont cependant l'inconvénient d'être nettement plus coûteuses. Par ailleurs, la limite de la rapidité du fonctionnement watchdog WDP, dans le cas de noeuds consécutifs, dépend essentiellement de la bande passante de la mémoire contenant le programme de vérification. L'implantation de ces méthodes rapides demande donc également l'utilisation de mémoires à accès rapide.

2.6.2.4.2. Exemple de singularités consécutives

Dans l'exemple de la figure 2.11 , le programme comporte deux instructions de séquencement conditionnelles Br-1 et Br-2 séparées de moins de $\alpha - \beta$ instructions (l'unité de programme est ici l'instruction, par souci de simplicité).

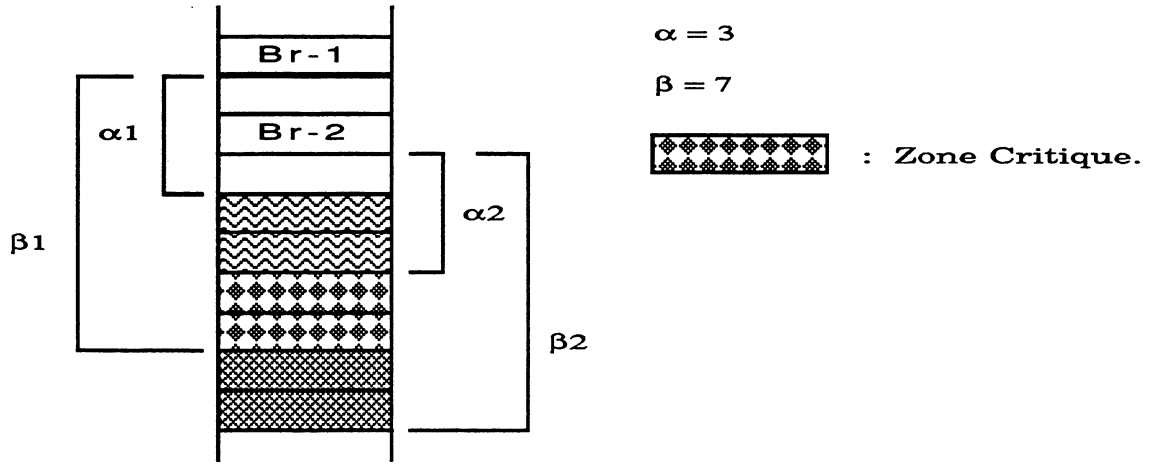


Figure 2.11 : exemple de branchements consécutifs dans la file d'attente du processeur.

Trois cas de comportement du processeur peuvent se présenter suivant qu'une rupture de séquence intervient :

a/ pendant le fetch des instructions marquées . La rupture ne peut être attribuée qu'à l'exécution de Br-1.

b/ pendant le fetch des instructions marquées . La rupture peut provenir soit de l'exécution de Br-1, soit de l'exécution de Br-2.

c/ pendant le fetch des instructions marquées . La rupture ne peut être attribuée qu'à l'exécution de Br-2.

Le fonctionnement du watchdog est donc le suivant (Br-1 vient d'être chargé par le processeur) :

- Recherche du noeud destination de Br-1 et attente de α fetchs.

- Si une rupture de séquence intervient avant β fetch depuis le chargement de Br-1 alors :
    - Pendant le fetch des instructions marquées , si la destination du branchement est égale à l'adresse de localisation du noeud destination de Br-1, le watchdog traite ce noeud destination, sinon il s'agit d'une rupture de séquence inopinée.

- Pendant le fetch des instructions marquées  , si la destination du branchement est différente de l'adresse de localisation du noeud destination de Br-1, le watchdog traite le noeud correspondant à Br-2. Si la destination du branchement est égale à l'adresse de localisation du noeud destination de Br-2, le watchdog traite ce noeud destination, sinon il s'agit d'une rupture de séquence inopinée.

- Si une rupture de séquence intervient après β fetch depuis le chargement de Br-1 et après α fetch depuis le chargement de Br-2, c'est-à-dire pendant le fetch des instructions marquées  , alors :
  - Si la destination du branchement est égale à l'adresse de localisation du noeud destination de Br-2, le watchdog traite ce noeud destination.
  - Si la destination du branchement est différente de l'adresse de localisation du noeud destination de Br-2, le watchdog considère qu'il s'agit d'une rupture de séquence inopinée.

- Si aucune rupture de séquence n'intervient après β fetch depuis le chargement de Br-2, alors le watchdog traite le noeud suivant Br-2.

2.6.2.4.3. Pipelines asynchrones et détection d'erreurs

Le fonctionnement particulier du processeur watchdog, dans le cas d'un pipeline asynchrone, accroît les risques de masquages d'erreur. En effet, le test de la signature, au niveau d'un noeud de séquencement, se fait indirectement à travers la comparaison des adresses de branchement et de localisation du noeud destination obtenu après hachage inverse du PCW avec la signature. Le fait de comparer l'adresse destination d'un branchement avec une liste de destinations, dans le cas où plusieurs branchements peuvent être à l'origine d'une rupture de séquence, entraîne donc une possibilité de masquage supplémentaire. Les taux de détection obtenus en section 2.4 doivent donc être divisés par le nombre γ (définition 2.7, cf 2.6.2.1) dans le cas d'un pipeline asynchrone.

Un autre problème lié à la comparaison des adresses de branchement avec une liste est le cas où les bits de poids faible des adresses de plusieurs destinations sont identiques, les bits de poids forts étant différents. Du fait de la localisation par adresses réduites, le watchdog n'est pas en mesure de connaître l'instruction à l'origine de la rupture de séquence. Il considérera que la première instruction chargée est la bonne, ce qui n'est pas forcément vrai et ceci sera détecté comme un

cas d'erreur, même si le comportement du processeur est parfaitement juste. Il ne s'agit donc pas d'un cas de masquage mais d'un cas de détection parasite.

Un tel événement est prévisible lors de la génération du programme de vérification. Une borne supérieure de la probabilité d'apparition d'une telle situation est égale à :

$$P_{\text{détection parasite}} = (\gamma - 1)\left(\frac{2^{\alpha-\rho} - 1}{2^{\alpha}}\right)$$

En prenant $\alpha = 24$, $\rho = 12$, $\gamma = 8$, on obtient : $P_{\text{détection parasite}} = 0{,}0017$.

2.6.3. Traitement des Exceptions

Le cas des départs en exception figure parmi les caractéristiques gênantes d'un processeur pour la vérification de son flot de contrôle par analyse de signature. En effet, il faut assurer l'invariance de la signature au départ et au retour d'une exception. Il faut donc considérer trois problèmes distincts :

- la reconnaissance d'un départ en exception,
- les traitements du moniteur ou watchdog au moment du départ,
- les traitements du moniteur ou watchdog au moment du retour[1].

Ces trois cas sont détaillés dans les sections suivantes.

2.6.3.1. Reconnaissance d'une exception

La reconnaissance d'un départ en exception peut en général se faire grâce aux informations placées par le processeur sur ses signaux externes au moment du départ en exception (demande de cycle de vectorisation, accès à une table de vecteurs), mais ceci est différent dans chaque cas de processeur vérifié.

Par ailleurs, certains processeurs ne signalent pas toutes leurs exceptions logicielles, en particulier lorsque celles-ci sont de type "autovectorisé", en général sur défaut logiciel pendant le traitement d'une instruction (division par zéro entre autres).

De même, pour certains processeurs, l'adresse de la table des vecteurs d'exception n'est pas fixe (processeurs avec MMU interne par exemple, Intel 80386 entre autres) et la détection d'un accès à cette table n'est donc pas réellement envisageable pour la reconnaissance des départs en exceptions.

---

[1] Un retour d'exception est une singularité qui peut être repérée de la même manière que toutes les autres et sa reconnaissance ne pose donc pas de problème, à la différence du départ en exception.

Il faut cependant remarquer que, par rapport au flot de contrôle d'un processeur, le cas d'un départ en exception correspond en fait à une erreur de séquencement (rupture de séquence inopinée ou branchement erroné).

Pour les méthodes qui sont capables de détecter instantanément les erreurs de séquencement (PSAb, OSLC, WDP), la reconnaissance d'un départ en exception ne pose donc pas de problèmes insurmontables. Il faut simplement que le moniteur ou le watchdog soit capable de différencier un départ en exception d'une erreur de séquencement. Ceci peut être fait par un marquage spécial de la première instruction de la routine d'exception, ou, moyennant une augmentation de la latence de détection, par un acquittement spécifique du processeur, programmé dans la routine d'exception.

Cette dernière solution a été choisie pour WDP de manière à respecter au maximum la généralité de la méthode, quel que soit le processeur vérifié.

2.6.3.2. Traitements associés au départ

Une fois l'exception reconnue, le dispositif de vérification doit empiler la signature courante, pour être en mesure de reprendre le calcul de la signature lors du retour de l'exception. Puis le dispositif de compaction doit être réinitialisé afin de pouvoir vérifier le flot de contrôle du processeur pendant le déroulement de la routine d'exception. Le traitement associé à un départ en exception ne pose en fait un problème que dans le cas d'un processeur pipeline (cf 2.6.2).

En effet, dans le cas d'un processeur pipeline, la signature empilée doit être celle de la dernière instruction exécutée et non pas celle correspondant au dernier chargement d'une unité de programme. Le moniteur ou le watchdog doit donc mémoriser toutes les signatures intermédiaires des instructions présentes dans la file d'attente du processeur. Ceci nécessite une file de signature de longueur égale à β .

Dans le cas d'un pipeline synchrone, la signature empilée est toujours la dernière de la file des signatures (signature d'age $\alpha = \beta$).

Dans le cas d'un pipeline asynchrone, le moniteur ou le watchdog n'est pas capable de connaître exactement l'age de la dernière instruction exécutée et il ne peut donc déterminer la signature devant être empilée. Pour remédier à ce problème, il existe deux solutions (au moins) :

- première solution : le moniteur ou le watchdog empile toutes les signatures d'age compris entre α et β , avec l'adresse microprocesseur de la dernière unité de programme chargée (donc compactée),

- deuxième solution : le moniteur ou le watchdog conserve momentanément la file de signatures intacte. L'age de la signature à empiler est alors déterminé par le microprocesseur (par logiciel) puis communiqué au dispositif de vérification qui empile alors la signature correspondante.

Dans le premier cas, il faut que le dispositif de vérification dispose d'une pile de taille importante et du temps nécessaire à l'écriture de toutes les informations à empiler.

Dans le second cas, la taille de la pile peut être diminuée mais ce mode de fonctionnement nécessite des aménagements logiciels du système vérifié et ne fonctionne pas pour les exceptions imbriquées trop rapprochées.

Dans le cas d'une méthode DSM telle que WDP, il faut par ailleurs assurer la cohérence du pointeur sur le programme de vérification, que le processeur dispose ou non d'un pipeline. Ceci est réalisé en plaçant le pointeur dans la pile au moment du départ. Après quoi, il faut que le watchdog soit en mesure de déterminer l'adresse du début de la routine d'exception dans son programme de vérification.

Pour un processeur pipeline, un problème supplémentaire se pose pour WDP : en cas de départ en exception sur une instruction de séquencement conditionnelle, le branchement peut avoir eu lieu sans que l'instruction destination ait été chargée. L'adresse de retour du processeur sera donc la destination du branchement et non pas celle située en séquence après la dernière unité chargée avant le départ.

Pour identifier ce cas gênant lors du retour de l'exception, la solution la plus simple consiste à empiler l'age du noeud en cours de traitement au moment du départ. De cette manière, le traitement du noeud interrompu peut reprendre normalement, en fonction de l'age du noeud, lors du retour d'exception.

2.6.3.3. Traitements associés au retour d'exception

Les traitements effectués par le dispositif de vérification au niveau d'un retour d'exception sont assez simples, en comparaison du cas d'un départ.

La reconnaissance ne pose aucun problème et le rôle du dispositif de vérification est donc simplement de tester la signature de la routine d'exception (avec la valeur associée à cette singularité) puis de recharger le dispositif de compaction avec la signature empilée lors du départ.

Dans le cas d'une méthode DSM, outre le traitement de la signature, il faut également restaurer le pointeur sur le programme de vérification (excepté pour OSLC).

Dans le cas de WDP pour la vérification d'un processeur pipeline, il faut également restaurer l'age du noeud interrompu, et dans le cas où une singularité était présente dans la file du processeur au moment du départ en exception, il faut traiter cette singularité, en fonction de son age, c'est-à-dire tout à fait normalement.

Dans le cas critique où une instruction de séquencement conditionnelle était présente dans le pipeline à un niveau d'exécution potentielle ($\alpha \leq age \leq \beta$), il faut faire la comparaison entre l'age dépilé pour le noeud à traiter et l'age réel de ce noeud calculé avec l'adresse de retour du processeur. En cas de différence, un branchement a eu lieu avant le départ en exception sans que l'instruction destination ait été chargée. Le traitement du noeud est alors identique à celui décrit en section 2.6.2.4 .

2.6.4. Conclusion

L'adaptation de la méthode WDP à différentes caractéristiques du processeur vérifié a été étudiée. Il ressort de cette analyse que la méthode WDP est compatible avec toutes les caractéristiques gênantes des processeurs pouvant être vérifiés par signature dérivée, au prix d'aménagements plus ou moins complexes, mais dont la plupart ne sont pas spécifiques à WDP. En effet, bon nombre de ces aménagements doivent également être faits pour les autres méthodes de vérification.

L'avantage de WDP, à ce niveau, est la complexité naturelle de son watchdog : le surcoût, entraîné par l'adaptation de la méthode aux caractéristiques gênantes du processeur, est ainsi quasiment négligeable pour WDP alors que pour d'autres méthodes, il peut dépasser, et de très loin, la complexité du dispositif original. En particulier, pour toutes les méthodes ESM, la nécessité d'une pile de signature pour les exceptions augmente sérieusement la complexité matérielle [Delo 91]. A l'inverse, les méthodes DSM disposent de base d'une mémoire locale et toutes nécessitent déjà une pile pour les sous-programmes (excepté OSLC). L'utilisation de cette même pile pour les exceptions n'entraîne donc aucun surcoût matériel.

Par ailleurs la prise en compte des caractéristiques du microprocesseur gênantes pour la génération de signature dérivée (lecture anticipée des instructions, exceptions) n'est pas toujours possible pour toutes les méthodes. En ce sens, WDP est une méthode dont la mise en œuvre est particulièrement générale, quel que soit le processeur vérifié.

2.7. Implantation d'un processeur watchdog WDP

L'étude théorique de WDP a donné lieu à trois réalisations, à savoir la conception de deux processeurs watchdog et le développement d'un modèle (écrit en langage VHDL) de watchdog WDP.

Après une définition assez générale, en section 2.7.1, de l'architecture permettant une adaptation simple à différents microprocesseurs, une description plus détaillée est donnée pour chacun des processeurs étudiés.

Le premier watchdog a été conçu pour fonctionner avec un 80386sx, mais toutes les adaptations nécessaires aux traitements du pipeline n'ont pas été implantées. Ce watchdog sera décrit brièvement en section 2.7.2 .

Un second watchdog, plus complet, a été conçu pour fonctionner avec un microprocesseur MC68000. Certaines de ses caractéristiques seront décrites en section 2.7.3 .

Un modèle VHDL de ce watchdog pour MC68000 et les résultats des simulations de ce modèle seront décrits en section 2.7.4 .

2.7.1. Architecture Modulaire

Afin d'être adaptable le plus facilement possible à de nombreux microprocesseurs, le processeur watchdog doit être conçu de manière modulaire en séparant le mieux possible ses différentes fonctions. L'architecture retenue est une architecture à quatre modules communiquants (Figure 2.12).

2.7.1.1. Module Interface Microprocesseur

Ce module est chargé du repérage du watchdog par rapport au programme du microprocesseur. Il doit donc détecter les ruptures de séquencement du microprocesseur afin d'en avertir la partie exécutrice. Pour cela, il vérifie que l'adresse microprocesseur de chargement des instructions est bien incrémentée à chaque fetch. Il doit également signaler l'occurrence d'un noeud. Pour cela il compare à chaque fetch l'adresse microprocesseur avec l'adresse du prochain noeud qui lui est fournie par la partie exécutrice (champ localisation de l'instruction watchdog). Ce module doit également calculer l'age d'un noeud, dans le cas d'un processeur pipeline.

Le nombre de bits de ce module dépend du nombre de bits de l'adresse microprocesseur stockée dans les instructions WDP (adresse réduite), et du nombre de bits du bus adresses du microprocesseur. Par exemple, si la largeur du bus d'adresses est de 32 bits et que les adresses réduites sont sur 12 bits, la partie du module interface microprocesseur qui s'occupe de la détection des noeuds sera sur 12 bits, tandis que la partie qui s'occupe de la détection des ruptures de séquence sera sur 32 bits.

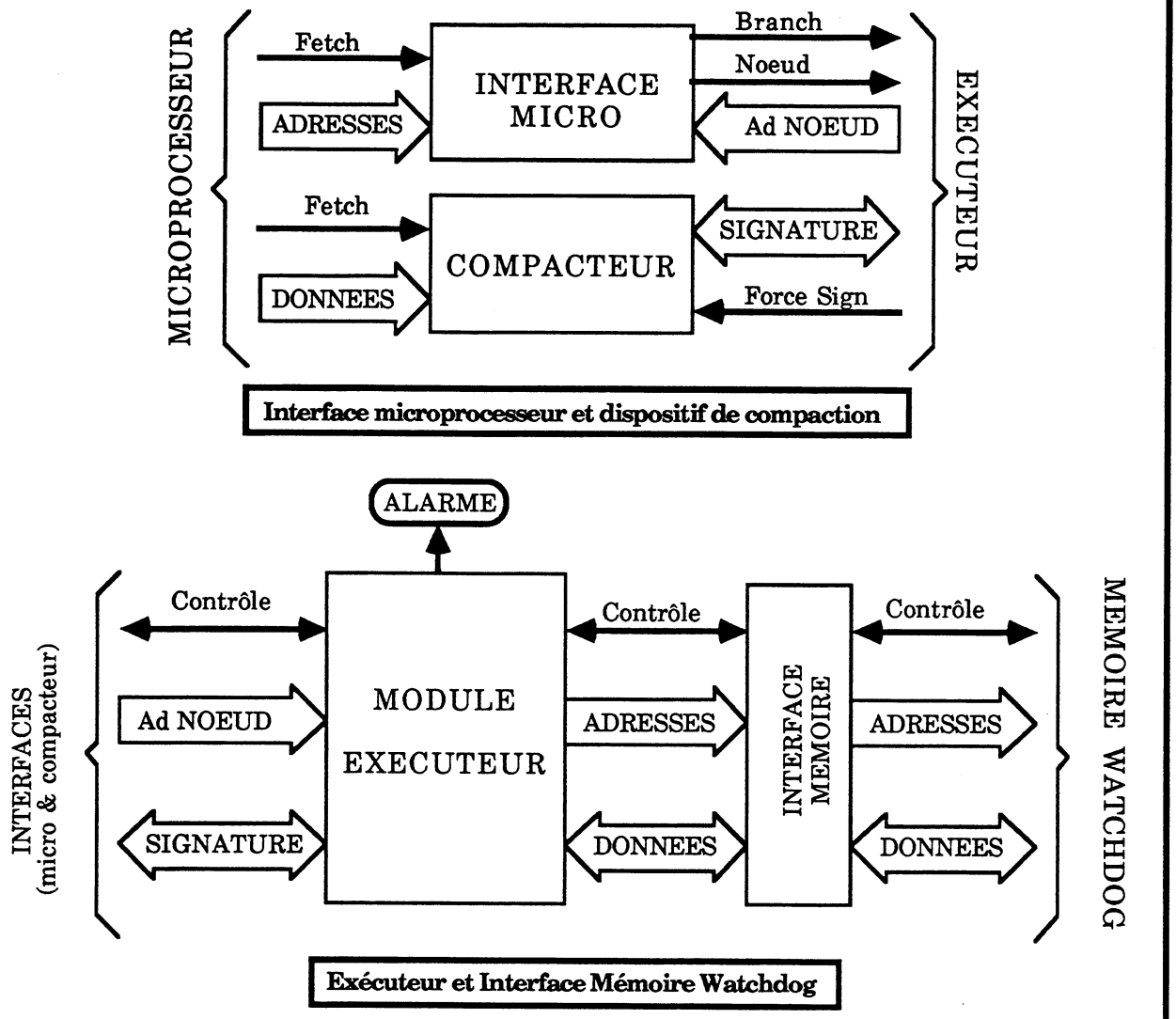


Figure 2.12 : architecture générale d'un processeur watchdog.

2.7.1.2. Module Compacteur

Ce module est chargé de la génération de la signature. A chaque fetch la valeur présente sur le bus de données du microprocesseur est compactée (a priori par un MISR). Le registre de signature peut être lu et écrit par le module exécuteur. Lorsque le watchdog prend en compte un pipeline synchrone et les exceptions, ce module contient une liste de signatures de taille β (Définition 2.6, cf 2.6.2.1), ceci afin de pouvoir conserver les signatures intermédiaires des instructions lues mais non encore exécutées. Le nombre de bits de ce module est fonction de la taille du bus de données du processeur vérifié, mais surtout de la largeur du bus d'adresses du processeur watchdog lui-même (à cause du hachage).

2.7.1.3. Module Exécuteur

Ce module est chargé d'exécuter le jeu d'instructions du watchdog. Il ne dialogue pas directement avec le microprocesseur mais uniquement avec les trois autres modules constituant le watchdog. C'est lui qui dirige les autres modules en fonction des instructions à exécuter. La description algorithmique du traitement à effectuer pour chaque type de noeud est faite en Annexe 1 . L'adaptation du fonctionnement de ce module au cas des processeurs pipelines a été décrite en section 2.6.2.4 .

Le fonctionnement de la partie exécutrice reste inchangé pour tous les microprocesseurs faisant partie de la même catégorie. On différencie principalement deux catégories : les microprocesseurs à pipeline synchrone et les microprocesseurs à pipeline asynchrone. Un microprocesseur sans pipeline fait partie de la première catégorie.

Le nombre de bits de ce module est à choisir en fonction de la taille maximum du programme watchdog (nombre de bits du PCW) et du format en mémoire des instructions. Pour les microprocesseurs 32 bits actuels, un format 16 bits semble toutefois satisfaisant.

2.7.1.4. Module Interface Mémoire

Ce module est chargé de gérer les échanges avec la mémoire, à savoir charger les instructions WDP ou empiler telle ou telle information. Cette interface dépend de l'architecture choisie au niveau du système complet. On peut citer au moins trois architectures possibles :

- mémoires séparées pour le watchdog et le processeur vérifié,
- utilisation de la mémoire du processeur par le watchdog avec prise de bus,
- intégration du watchdog à un contrôleur de cache pour le microprocesseur : la mémoire du watchdog est la mémoire principale du microprocesseur. Il n'y a donc pas de prise de bus du microprocesseur par le watchdog qui doit cependant gérer ses conflits d'accès avec le cache.

La première solution est la plus simple à implanter mais c'est celle qui impose le nombre de composants supplémentaires le plus élevé.

La seconde solution est moins coûteuse, en nombre de composants, que la première, mais le watchdog doit prendre le bus du microprocesseur et le gérer en gênant le moins possible le microprocesseur : c'est la plus coûteuse en performances.

La troisième solution est particulièrement intéressante parce qu'en s'intégrant au cache instruction, le watchdog n'entraîne aucun composant supplémentaire tout en respectant les performances du système.

2.7.2. Application au microprocesseur Intel 80386sx

Un premier essai de réalisation d'un processeur watchdog a eu lieu en 1990 pour le cas du microprocesseur Intel 80386sx. Ce processeur [Mich 90] implante exactement la description du fonctionnement du watchdog telle qu'elle est faite en Annexe 1 . Dans ce cas, le traitement des noeuds se fait toujours, au moment du chargement par le processeur de la singularité correspondant à un noeud. Le watchdog dispose par ailleurs d'une lecture anticipée de ses propres instructions. Pour le traitement des exceptions, le watchdog dispose d'une file de trois signatures intermédiaires.

Au moment de la réalisation de ce premier processeur watchdog, les problèmes liés aux pipelines et au traitement des exceptions n'avaient pas encore été clairement identifiés (ils l'ont été en partie grâce à cette réalisation). Cela signifie que ce processeur est limité au cas des programmes d'application dans lesquels plusieurs singularités ne peuvent se trouver simultanément dans le pipeline du 80386sx. Le but de cette application était double : d'abord vérifier que la méthode WDP n'était pas incompatible avec une réalisation pratique, ensuite avoir un ordre d'idée du coût matériel.

2.7.2.1. Caractéristiques du microprocesseur Intel 80386sx

Le 80386sx Intel a la même architecture 32 bits que le 80386 (dx) mais la taille des bus est réduite à 16 bits pour le bus de données et 24 bits pour le bus d'adresses. Il dispose d'une MMU intégrée qui autorise la pagination mémoire et quatre niveaux de protection. La capacité d'adressage physique est de 16 Moctets et la taille d'une page est 4K octets (2^{12} octets). Trois signaux sont utilisés pour la définition des cycles bus (D/C, M/IO, W/R) . Ils permettent de savoir facilement lorsque le microprocesseur effectue un cycle d'accès au code de son programme ou un cycle de reconnaissance d'interruption. Ceci facilite beaucoup la génération de la signature. Un signal "READY" permet d'allonger la durée d'un cycle bus. Ce signal est contrôlé par le watchdog pour ralentir le microprocesseur dans certains cas.

2.7.2.2. Implantation du processeur watchdog

L'architecture de ce watchdog est calquée sur celle de la figure 2.12 . Les modules d'exécution, de compaction et d'interface mémoire sont sur 16 bits.

Le module d'interface avec le microprocesseur est sur 12 bits pour la partie chargée de la détection des noeuds (utilisation d'adresses réduites du fait de la

pagination) et sur 24 bits pour la partie chargée de la détection des ruptures de séquence dans les fetch. Pour la prise en compte des exceptions, il avait été prévu de permettre au microprocesseur un accès direct aux registres du watchdog.

La partie exécutrice comprend un registre de code opération (RCOP), un compteur ordinal (PCW), un pointeur de pile (SPW) plus de la logique (portes OU Exclusif, comparateur, incrémenteur).

La partie interface mémoire intègre un mécanisme de prefetch des instructions watchdog, permettant de fournir l'adresse du prochain noeud "au plus tôt" à la partie interface microprocesseur.

Le module compacteur dispose d'un MISR programmable. La partie interface microprocesseur possède un registre "adresse microprocesseur" sur 24 bits et un registre "adresse du prochain noeud" sur 12 bits.

Le processeur watchdog fonctionne avec une mémoire locale organisée en mots de 16 bits et séparée de la mémoire centrale du microprocesseur.

Tableau 2.8 : complexité du watchdog WDP pour un 80386sx Intel.

Module	Bits	Nbre de portes	Pourcentage
Exécution	16	2293	51,1%
Interface microprocesseur	12	1156	25,8%
Compaction	16	841	18,8%
Interface mémoire	16	191	4,3%
Total		4481	100%

La complexité des différents modules est résumée dans le tableau 2.8 . On peut remarquer que le nombre total de portes est suffisamment faible pour intégrer le tout économiquement dans un circuit prédiffusé ou un FPGA (Programmable Gate Array). Le circuit nécessite 84 plots d'entrées/sorties. Le coût global au niveau d'une carte est donc assez réduit.

Ce résultat encourageant a donné lieu à une deuxième réalisation, beaucoup plus complète, pour le cas d'un processeur MC68000.

2.7.3. Application au microprocesseur MC68000

Le but de cette application est de réaliser un processeur watchdog aussi complet que possible pour un processeur disposant d'un pipeline.

La raison du choix du microprocesseur MC68000 est triple :

- son mécanisme de prefetch des instructions constitue un pipeline synchrone à deux étages tout à fait typique,

- les informations concernant son fonctionnement interne sont relativement plus disponibles que pour d'autres microprocesseurs,
- son usage est particulièrement répandu et facilite les développements éventuels.

Après un rapide rappel des caractéristiques du microprocesseur MC68000, nous verrons les techniques utilisées ici pour la prise en compte du mécanisme de prefetch et des exceptions, dans le cas réel du MC68000. Une description complète du watchdog se trouve en Annexe 3 .

2.7.3.1. Caractéristiques du MC68000

Le MC68000 est un microprocesseur 16/32 bits, c'est-à-dire qu'il possède une architecture interne sur 32 bits mais un bus de données sur 16 bits seulement. En cela il se rapproche tout à fait d'un 80386sx, ce qui a permis de reprendre globalement l'étude précédente, en modifiant l'interface microprocesseur. Le format des instructions watchdog et des différents modules sont rigoureusement identiques au cas du 80386sx .

Le MC68000 dispose également de signaux (FC0, FC1, FC2) permettant d'identifier les accès au code du programme et les cycles de reconnaissance d'interruptions. Un signal "DTACK" d'introduire des cycles d'attente pendant les accès mémoire (échanges asynchrones). Ce signal est donc là encore contrôlé par le watchdog.

2.7.3.2. Prise en compte du pipeline

Le MC68000 possède un mécanisme systématique de lecture anticipée des instructions d'un mot mémoire. Pour cela, il dispose d'un registre de prefetch et d'un registre de décodage des instructions sur 16 bits. Pour être décodé et exécuté, le mot de code opératoire de l'instruction doit forcément passer par le registre de prefetch. Pendant que le mot de code est en cours de décodage, le mot suivant est chargé dans le registre de prefetch. Lorsqu'une instruction de branchement est exécutée et que le branchement est pris, les registres de décodage et de prefetch sont vidés. L'exécution de la prochaine instruction n'aura donc lieu que deux fetchs plus tard. Selon les définitions données en section 2.6.2.1 , le mécanisme de prefetch du MC68000 constitue un pipeline synchrone d'une longueur $\beta = 2$, avec $\alpha = 1$, l'unité de programme étant le mot de 16 bits (et donc pas forcément une instruction complète).

La solution implantée dans le watchdog est celle décrite en section 2.6.2.4 . Elle implique donc que le watchdog traite les singularités non plus au moment de leur chargement par le processeur mais à retardement, donc en fonction de leur age.

Un problème pratique se pose, à cause de la localisation des noeuds par adresse réduite (12 bits de poids faible seulement). Par analogie au cas d'un processeur disposant d'une MMU intégrée, on appellera ici "page" une zone d'adressage sur 12 bits.

Selon que le noeud à traiter et l'instruction courante du microprocesseur se trouvent ou non sur la même page, le résultat de la soustraction pour le calcul de l'age est différent (les frontières de "page" se situent aux adresses microprocesseur dont les 12 bits de poids faible sont à 0).

Lors de la soustraction entre les bits de poids faible de l'adresse courante du microprocesseur et le champ localisation du noeud à traiter, pour déterminer l'age de ce noeud, un résultat positif est censé indiquer que le noeud est déjà dans le pipeline et un résultat négatif que le microprocesseur ne l'a pas encore atteint. Ceci est vrai dans le cas où l'adresse microprocesseur courante est dans la même page que le noeud à traiter. Dans le cas contraire, le résultat de la soustraction doit en fait être inversé.

Appelons N1 le noeud dont on vient juste de terminer le traitement, et N2 le prochain noeud à traiter. Afin de pouvoir vérifier si le microprocesseur poursuit bien ses fetch en séquence, le watchdog dispose d'un registre de α bits (où α est la largeur du bus d'adresses) et d'un incrémenteur. Si le watchdog utilise des adresses réduites de 12 bits, il y a changement de "page", lorsque la treizième retenue de l'incrémenteur passe à 1 . Appelons Ret13 cette retenue. Notons @N1 l'adresse microprocesseur réduite de N1 et @N2 l'adresse microprocesseur réduite de N2. Notons @M les bits de poids faible de l'adresse microprocesseur courante de chargement des instructions.

La page sur laquelle se trouve N2 se détermine ainsi :

Si @N2>@N1 et Ret13 = 1 : @N2 et @M pointent des pages différentes, Ret13 doit être remis à 0 et N2 est déjà dans le pipeline,

Si @N2>@N1 et Ret13 = 0 : @N2 et @M pointent la même page,

Si @N2≤@N1 et Ret13 = 1 : Ret13 doit être remis à 0 et @N2 et @M pointent la même page,

Si @N2≤@N1 et Ret13 = 0 : @N2 et @M pointent des pages différentes, et N2 n'a pas encore été atteint par le microprocesseur.

Ceci signifie que pour savoir si N2 et l'instruction pointée par l'adresse microprocesseur courante sont sur la même page, il faut effectuer la soustraction @N2-@N1 et connaître Ret13. Aussi, pour savoir si N2 est déjà dans le pipeline du microprocesseur, le watchdog effectue la soustraction @N2-@N1 et vérifie si Ret13

est passée à 1 depuis le fetch de N1. En fonction des résultats de ces opérations, il peut interpréter correctement le résultat de la soustraction @M-@N2.

Ce problème est illustré par un exemple sur la figure 2.13 . Dans cet exemple, la taille d'une page est de 32 mots (codage sur 5 bits) seulement ce qui n'enlève rien à la généralité des explications.

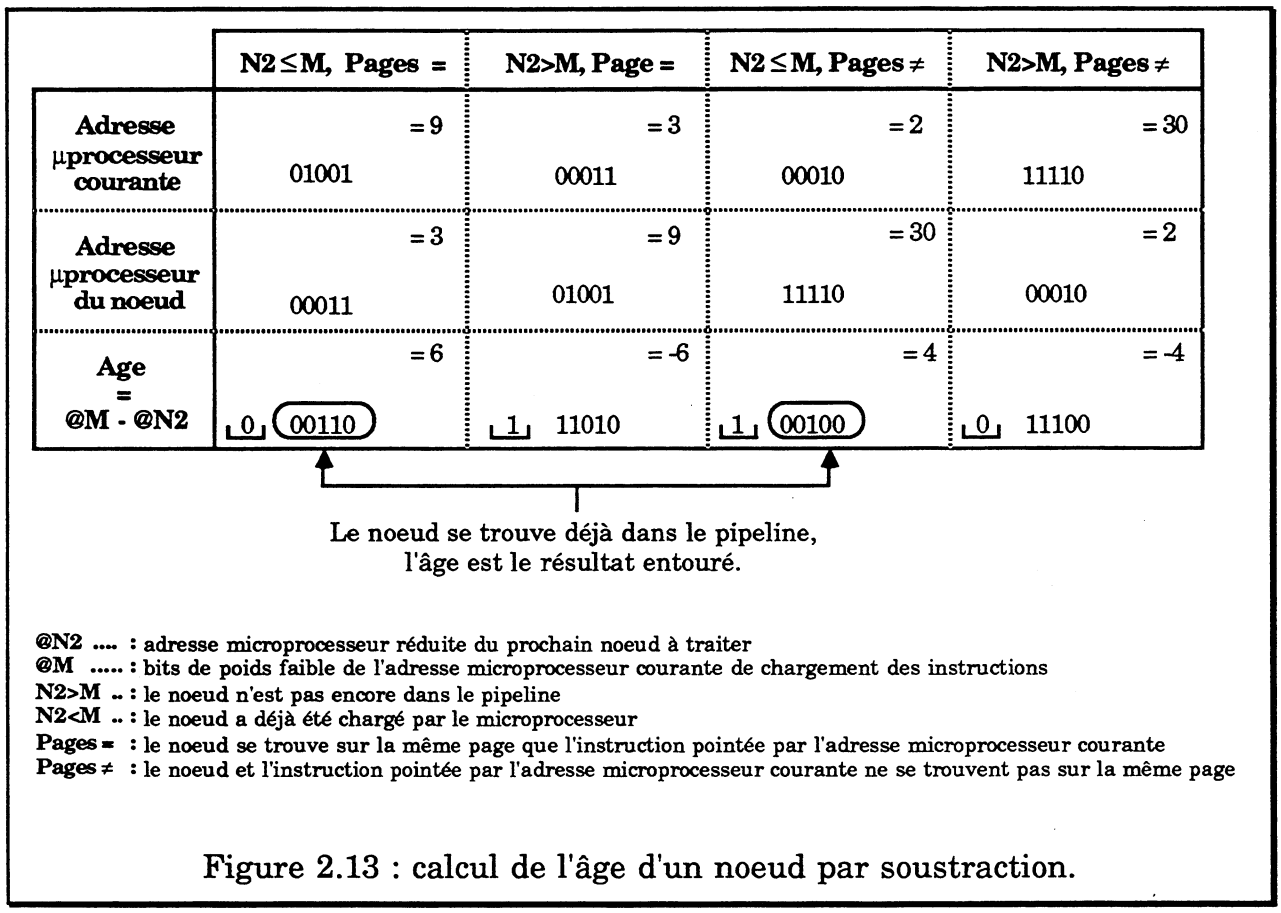


Figure 2.13 : calcul de l'âge d'un noeud par soustraction.

2.7.3.3. Prise en compte des exceptions

Dans le cas du MC68000, la méthode utilisée pour la reconnaissance d'un départ en exception est la détection des ruptures de séquences inopinées. Pour différencier un départ en exception d'une authentique erreur de séquencement, la convention utilisée est de marquer le début d'une routine d'exception avec une instruction spéciale. Le rôle de cette instruction est également de fournir au watchdog un pointeur sur le début du programme de vérification associé à la routine d'exception.

L'instruction de marquage d'une routine d'exception est un branchement dont l'adresse de destination est située deux mots après l'instruction elle-même (Figure 2.14). Ainsi, avec le prefetch du MC68000, le programme, au niveau du branchement , sera lu séquentiellement. Le mot situé immédiatement après le

branchement ne sera pas exécuté mais il aura tout de même été lu par le processeur. Ce mot est le pointeur attendu par le watchdog. L'adresse ainsi obtenue doit correspondre à un noeud d'initialisation dans le programme de vérification.

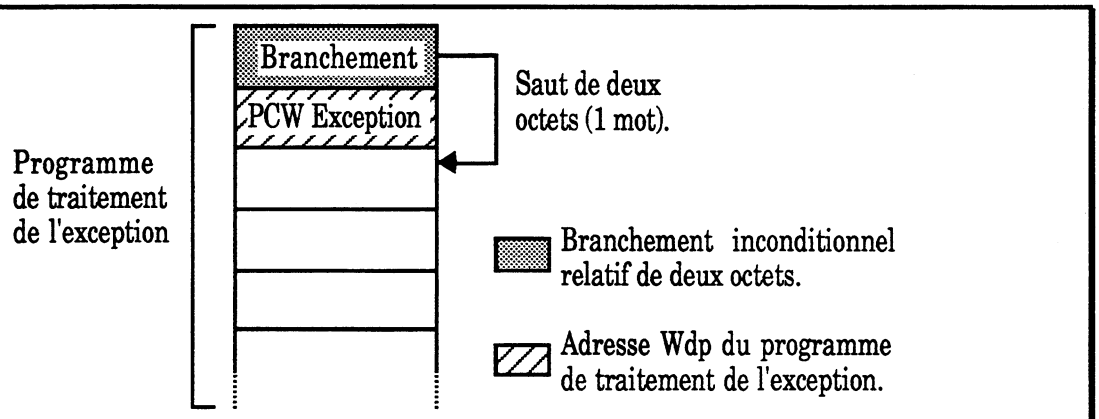


Figure 2.14 : reconnaissance d'un début de routine d'exception.

2.7.4. Modélisation et simulation d'un watchdog WDP pour MC68000

Le processeur watchdog WDP décrit en section 2.7.3 a été modélisé en VHDL de manière à pouvoir simuler le comportement d'un système complet, avec microprocesseur MC68000, processeur watchdog WDP, et leurs mémoires respectives, contenant différents exemples de programmes.

Le modèle du watchdog figure en Annexe 3.5 .

Le premier intérêt de ces simulations a été de valider le logiciel de génération du programme de vérification pour le watchdog.

Le deuxième et principal intérêt de ces simulations au niveau système est d'étudier le comportement réel du watchdog lorsque des combinaisons "aléatoires" d'événements surviennent. En effet, lors de la simulation d'un composant seul, la vérification ne porte que sur les événements définis par les vecteurs de simulation appliqués au composant. En général, les événements externes sont simulés individuellement, toutes les combinaisons possibles ne pouvant être représentées dans les vecteurs de simulation.

Dans une simulation système, un grand nombre de combinaisons d'événements apparaissent naturellement, sans qu'il soit nécessaire de les programmer explicitement. En particulier, le fonctionnement du watchdog en cas de succession de noeuds plus ou moins rapprochés peut facilement être observé, avec des exemples adéquats de programme exécuté par le microprocesseur. De même, des interruptions du microprocesseur peuvent être déclenchées d'une

manière aléatoire par un modèle spécifique. Ainsi, la plupart des cas critiques du traitement des exceptions peuvent être simulés facilement. Une adaptation du modèle permet d'enregistrer tous les cas qui ont été simulés. Les combinaisons qui ne sont pas apparues spontanément en simulation sont alors programmées explicitement.

Le comportement du watchdog a également ainsi été étudié avec des temps d'accès à sa mémoire locale plus où moins long, afin de voir dans quelle mesure il était possible d'avoir une mémoire watchdog plus lente que la mémoire du processeur vérifié et sans que le processeur vérifié soit arrêté trop souvent.

Il a été remarqué à ce sujet que la présence d'un pipeline, dans le processeur vérifié, favorise l'allongement des temps d'accès de la mémoire du watchdog. En effet, entre le moment du chargement d'une instruction de séquencement et son exécution effective par le processeur vérifié, le watchdog a le temps, en général, de charger le noeud destination et une partie (grâce à son propre mécanisme de prefetch) du noeud situé après la destination. Ceci permet de ne pas ralentir le processeur vérifié, le watchdog ayant les informations nécessaires pour le traitement des singularités.

De même, la possibilité pour le watchdog de traiter les singularités en fonction de leur age, permet, même dans le cas où le watchdog ne dispose pas à temps des informations pour le traitement d'une singularité, de ne pas ralentir le processeur. La limite au retard du watchdog est cependant définie par la longueur de la file des signatures intermédiaires dans le module compacteur. Un allongement de cette file, bien qu'inutile par rapport à la taille du pipeline, peut ainsi permettre de ralentir encore les accès mémoire du watchdog. Un compromis entre temps d'accès mémoire (donc ralentissement potentiel du processeur vérifié) et coût matériel du module compacteur (donc du watchdog) peut donc être défini en fonction des caractéristiques du système vérifié.

La conclusion de ces simulations est que le processeur watchdog de WDP fonctionne parfaitement dans le cas d'un système à base de MC68000.

2.7.5. Conclusion sur l'implantation de la méthode WDP

L'étude de la méthode WDP a donné lieu à deux réalisations pratiques, sous la forme de deux processeurs watchdog. Une conception modulaire de ces derniers a été préférée à une conception "monobloc", ceci permettant de pouvoir les adapter plus facilement à différents microprocesseurs.

La première réalisation a permis de quantifier le coût matériel de la méthode tout en vérifiant sa faisabilité. Le coût matériel étant tout à fait acceptable, ceci a conduit à approfondir l'étude de la méthode.

Une deuxième réalisation plus complète a ainsi permis d'étudier l'impact au niveau de la définition du processeur watchdog, de la prise en compte du pipeline et des exceptions du processeur. Il a été constaté que l'implantation générale du watchdog n'avait pas à être redéfinie.

Un modèle VHDL de cette réalisation a été simulé dans son environnement, ce qui a permis de bien mettre au point les différents aspects de la synchronisation du watchdog avec d'une part le processeur vérifié et d'autre part la mémoire locale du watchdog. La synchronisation inter-modules a également été optimisée de manière à ce que le watchdog puisse répondre rapidement à des combinaisons d'événements externes.

2.8. Conclusion sur la méthode WDP

La méthode WDP est une méthode de vérification de flot de contrôle, par signature dérivée, dont les avantages sont particulièrement intéressants.

Elle permet, comme toutes les méthodes DSM, de ne pas avoir à modifier le programme vérifié, ce qui permet d'assurer une compatibilité logicielle et de ne pas ralentir systématiquement le processeur. De plus la génération des informations nécessaires à la vérification est particulièrement simple et rapide, et peut (doit) être faite sur le code exécutable du programme vérifié, donc sans qu'il soit nécessaire de disposer d'une version spéciale de compilateur, contrairement aux méthodes ESM.

Au niveau de l'efficacité du test, on a vu que la méthode WDP était plus efficace que toutes les autres méthodes proposées jusqu'ici, et que par ailleurs, le calcul de son taux de détection théorique ne posait aucun problème, contrairement aux méthodes ESM avec ajustements.

L'implantation de la méthode WDP est certes plus coûteuse que certaines méthodes, en particulier ESM, surtout au niveau de la taille du programme de vérification et des modifications entraînées au niveau de l'architecture du système.

Mais l'intérêt principal de la méthode WDP est qu'elle peut s'utiliser, avec des aménagements minimes, quel que soit le processeur vérifié, à partir du moment où il est possible de générer une signature dérivée du programme exécuté (pas de mémoire cache intégrée au processeur). En particulier, la méthode WDP est la seule à supporter, quasiment sans restrictions, les processeurs disposant d'un pipeline asynchrone pour la lecture anticipée de leurs instructions.

Chapitre 3. DJAM : une méthode DSM avec ajustements

3.1. Introduction

DJAM, pour "Disjoint Justifying Arc Monitoring", est, comme son nom l'indique, une méthode DSM avec ajustements sur arcs. Il s'agit de la première méthode DSM assurant l'invariance de la signature par ajustements. Rappelons que les méthodes DSM proposées jusqu'à maintenant (Cerberus 16, OSLC, ASIS, WDP) sont toutes très différentes :

- Cerberus 16 et ASIS ne respectent pas l'invariance de la signature en tout point du programme,
- OSLC et WDP réinitialisent le dispositif de compaction pour assurer l'invariance de la signature.

Comme pour WDP, la méthode DJAM correspond au test en ligne d'un processeur par analyse de signature dérivée, les codes des instructions exécutées étant compactés par un dispositif tel que ceux décrits en section 1.2 .

La présentation de la méthode DJAM faite dans ce chapitre reprend globalement le plan utilisé pour WDP au chapitre précédent, à l'exception de la partie implantation. En effet, cette méthode a été implantée d'une manière très particulière sur un prototype d'unité centrale d'automate programmable, en collaboration avec la société Télémécanique, et ceci sera décrit au chapitre 4. La méthode en elle même, par contre, s'applique quel que soit le processeur vérifié et c'est la raison pour laquelle elle est présentée séparément.

Dans tout ce chapitre, de nombreuses comparaisons avec WDP seront faites, mais pour que la lecture du chapitre 2 concernant WDP ne soit pas indispensable à la compréhension de ce chapitre, un certain nombre de points seront volontairement répétés.

3.2. Description de la méthode

3.2.1. Origine de la méthode

La méthode DJAM est inspirée de WDP, dont elle reprend le principe de considérer toutes les instructions de branchement comme des singularités, ce qui permet d'assurer à la fois la gestion du pointeur de références et l'invariance de la signature.

La gestion du pointeur de références est assurée comme dans WDP en suivant la trace du processeur grâce à une détection des ruptures de séquences au niveau des instructions de séquencement.

L'invariance de la signature est assurée comme dans WDP sur les ruptures de séquences, mais la ressemblance s'arrête là, car dans WDP l'invariance est assurée par réinitialisation du dispositif de compaction, alors que pour DJAM elle est assurée par ajustement.

Par ailleurs, le repérage des singularités peut être fait par décodage d'instruction pour DJAM car les seules singularités devant être repérées pour assurer l'invariance de la signature sont les instructions de séquencement. Les points de jonction n'ont pas besoin d'être repérés, car il n'est pas nécessaire de tester la signature lors de l'arrivée séquentielle à leur niveau, la signature n'étant pas réinitialisée. Les singularités liées à l'invariance de la signature peuvent donc être complètement identifiées par décodage d'instruction (localisation et type), sans nécessiter d'informations supplémentaires dans les références. WDP, au contraire, utilise un repérage par les adresses.

On peut donc facilement présager que le coût mémoire minimal de DJAM sera largement inférieur à celui de WDP : d'une part, les noeuds destination n'existent plus, et d'autre part, les adresses des singularités n'ont pas à être stockées dans les références. Cette diminution du coût mémoire par rapport à WDP est d'ailleurs une des raisons d'être de cette méthode, car WDP présente de très bonnes caractéristiques par ailleurs.

L'ajustement lors des ruptures de séquence présente toutefois un inconvénient. Le pointeur de référence ne peut plus être haché avec l'information de signature de la singularité car cette information est une valeur d'ajustement qui doit être stockée de manière explicite[1], contrairement à une signature de référence qui peut être hachée moyennant une certaine probabilité de masquage. Il est donc nécessaire de stocker explicitement, pour chaque instruction de branchement, une valeur d'ajustement et un pointeur sur la référence destination.

Il faut remarquer que le dispositif matériel assurant le fonctionnement de la méthode DJAM n'est pas un watchdog au sens où il n'exécute pas un programme. Les informations de test ne contiennent en effet aucune instruction pour le fonctionnement du dispositif de test mais seulement des données traitées par un moniteur. Ceci justifie le terme "monitoring" dans le nom de la méthode.

---

1 On pourra remarquer le même phénomène entre SIS et Wil87 .

3.2.2. Principe de fonctionnement

La méthode DJAM reprend en fait exactement le principe du schéma d'invariance de [Wilk 87] (cf 1.4.4.2.1) dans sa version de base, c'est-à-dire que l'invariance de la signature est assurée par des ajustements placés sur tous les arcs de séquencement du programme. Par contre, puisqu'il s'agit ici d'une méthode DSM, les valeurs d'ajustement ne sont pas des paramètres des instructions de séquencement mais elles sont stockées séparément du programme et sont accédé grâce à un pointeur (RP). La gestion de ce pointeur nécessite le stockage d'informations sur la structure du graphe vérifié en plus des valeurs d'ajustement et de test. Ces informations sont placées avec les valeurs d'ajustement et de test dans une "table des références".

Le fonctionnement de l'analyseur pour les ajustements est assuré lors des ruptures de séquence suivant le principe de la figure 3.1 .

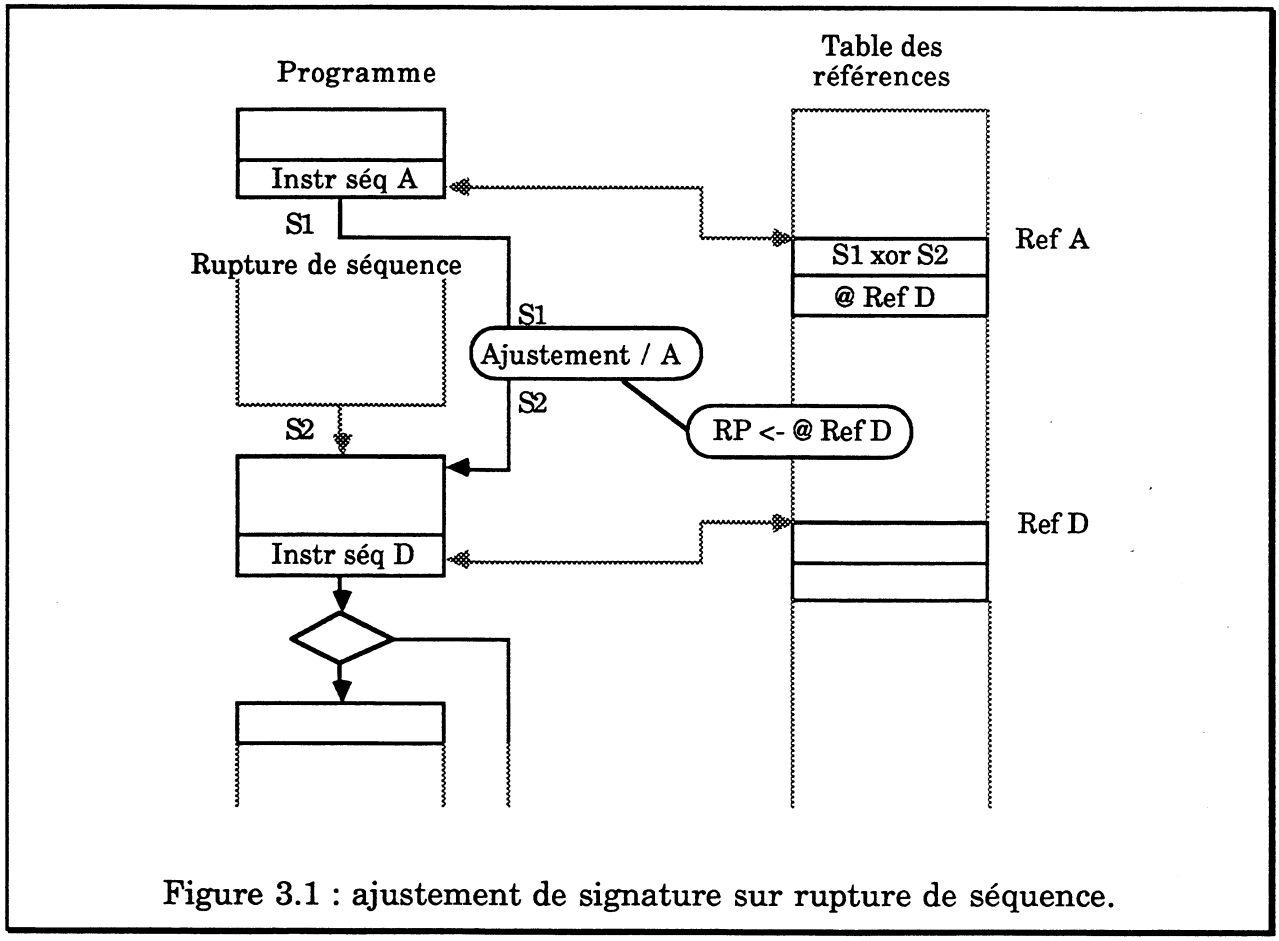


Figure 3.1 : ajustement de signature sur rupture de séquence.

A chaque instruction de séquencement (branchements, sauts, appels et retours de sous-programmes), correspondent donc une valeur d'ajustement et un pointeur sur la prochaine référence à accéder dans la table des références :

- la valeur d'ajustement est le OU Exclusif de la signature intermédiaire à l'instruction de séquencement et de la signature intermédiaire à l'instruction précédant la destination de l'arc de séquencement,
- le pointeur de prochaine référence est l'adresse de la référence qui correspond à la première référence en séquence après la destination de l'instruction de séquencement.

Pour les instructions de séquencement qui n'entraînent pas de rupture de séquence, l'ajustement n'a pas lieu. La référence est simplement ignorée, et le pointeur de référence incrémenté de deux mots, pour passer à la référence suivante qui est celle située immédiatement après dans la table des références (Figure 3.2).

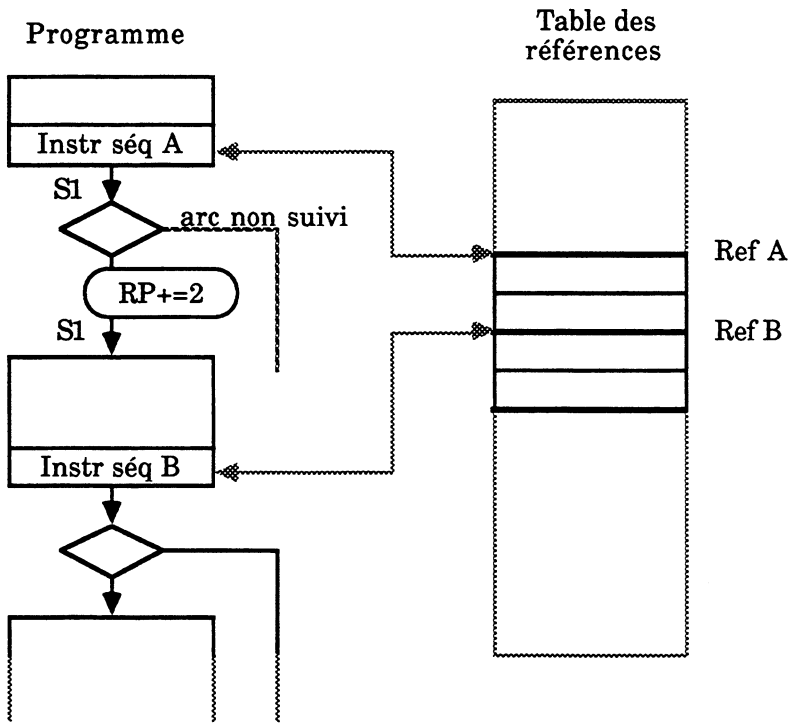


Figure 3.2 : instructions de séquencement sans rupture de séquence.

Il faut remarquer que, dans le schéma retenu, toutes les instructions de séquencement nécessitent un ajustement, alors qu'en théorie, seules celles dont la destination est un point de jonction devraient en avoir (cf 1.4.4.2.1). La raison vient du fait qu'il n'est pas possible de différencier les instructions de séquencement nécessitant un ajustement et celles qui n'en ont pas besoin car elles ont le même code d'instruction.

Il n'est évidemment pas possible d'utiliser des codes instructions différents pour discriminer les instructions de séquencement, comme cela est proposé dans [Wilk 87] car alors on modifie le programme vérifié et l'intérêt de l'implantation DSM est annulé. Par ailleurs, la gestion du pointeur de références impose que tous les branchements (conditionnels au moins) possèdent un pointeur de prochaine référence, même s'ils ne nécessitent pas d'ajustement.

Le nombre d'ajustements est ainsi plus élevé que le minimum théorique d'un schéma d'analyse de signature avec ajustements. Ceci augmente donc le coût mémoire mais par contre, la génération de la table des références est simplifiée car une bonne partie des signatures intermédiaires peuvent être choisies de manière arbitraire, ce qui simplifie le calcul des signatures intermédiaires et des valeurs d'ajustement (cf 1.4.4.2.4).

3.2.3. Invariance des sous-programmes

L'invariance des sous-programmes est assurée d'une manière un peu particulière pour faciliter l'application de la méthode dans le cas d'instructions de départ en sous-programme conditionnelles. L'instruction située immédiatement après un appel de sous-programme est alors un point de jonction (Figure 3.3). Il faut donc ajuster la signature sur un des deux chemins conduisant à cette instruction, c'est-à-dire soit sur retour de sous-programme, soit à l'instruction de départ si la condition est fausse.

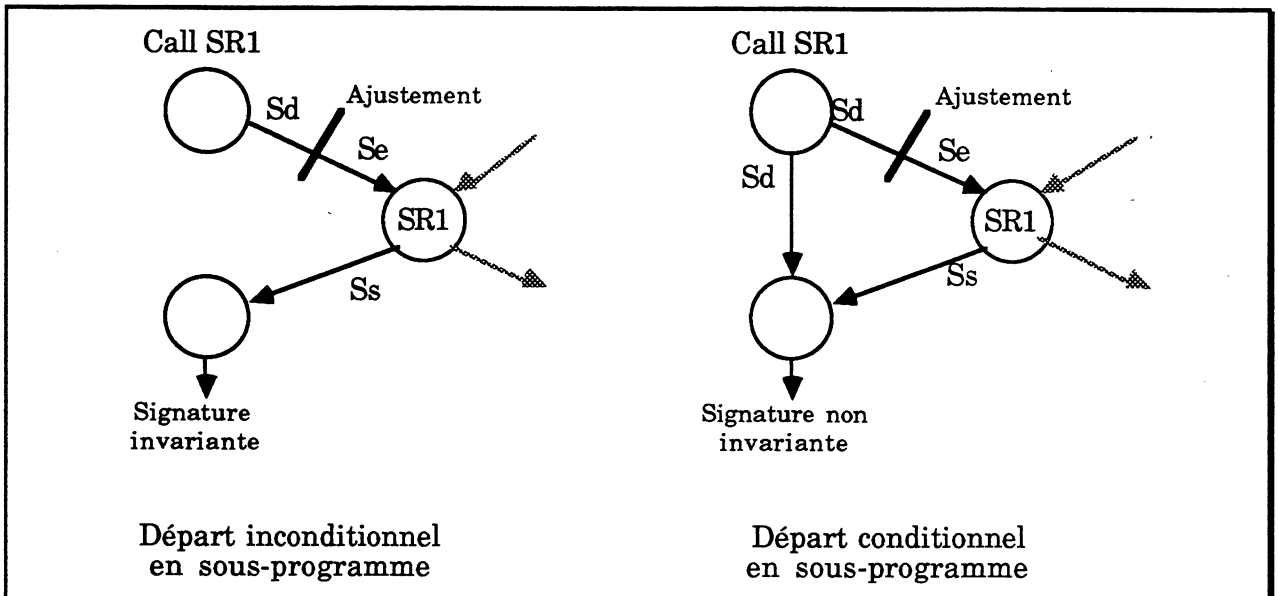


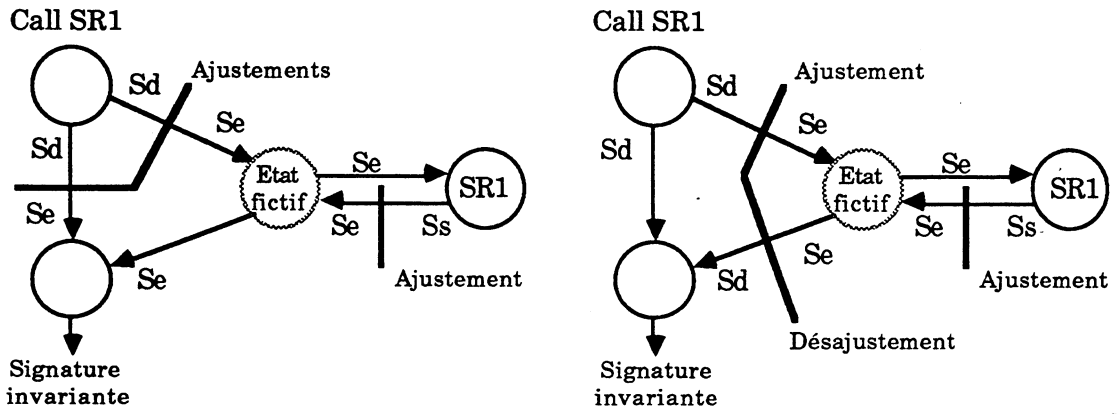
Figure 3.3 : problème de l'invariance des signatures pour les sous-programmes conditionnels.

Un ajustement sur retour de sous-programme n'est pas envisageable car toutes les instructions de départ appelant un même sous-programme n'ont pas la même signature intermédiaire et donc une valeur unique d'ajustement pour un retour de sous-programme ne suffit pas.

Un ajustement sur condition fausse nécessite deux valeurs d'ajustement. La première sert pour ajuster sur la valeur d'entrée du sous-programme si la condition d'appel est vraie et la seconde permet d'ajuster sur la valeur de sortie du sous-programme si la condition d'appel est fausse.

Pour éviter la présence de ces deux valeurs, la solution consiste à rendre la signature de sortie d'un sous-programme égale à la valeur de la signature en entrée de ce sous-programme, ce qui peut être fait avec une valeur d'ajustement pour chaque retour d'un même sous-programme.

Ainsi, l'ajustement avec condition fausse ou condition vraie à l'appel peut être fait avec la même valeur. Cependant, cette solution a été rejetée (Figure 3.4.a) car ce schéma d'invariance pour les sous-programmes complique sérieusement la génération des références. En effet, la signature après un départ de sous-programme dépend de la signature du sous-programme ce qui pose problème pour générer les références de manière modulaire et dans le cas de sous-programmes récursifs.



(a) signature dépendante du sous-programme

(b) signature indépendante du sous-programme

Figure 3.4 : invariance des sous-programmes.

La solution retenue (Figure 3.4.b) consiste en fait à "désajuster" la signature en retour de sous-programme après l'avoir ajustée sur la valeur d'entrée. Ce désajustement est fait avec la valeur d'ajustement d'appel du sous-programme grâce aux propriétés de la fonction utilisée pour les ajustements (OU Exclusif) :

si $S3 = S1 \oplus S2$
alors $S1 = S3 \oplus S2$
et $S2 = S3 \oplus S1$

Avec la solution retenue, l'invariance de la signature pour les départs en sous-programme conditionnels est assurée et la signature intermédiaire à l'instruction située immédiatement après un départ en sous-programme ne dépend pas de la signature du sous-programme. Les avantages sont donc les mêmes qu'avec le schéma de [Wilk 87] qui nécessite une mise en pile de la signature (cf 1.4.4.2.3) :

- la décorrélation des signatures intermédiaires est maximale,
- la génération des références est facilitée et peut être modulaire (génération indépendante pour le programme principal et les sous-programmes),
- la génération des références dans le cas de sous-programmes récursifs ne pose pas de problème particulier.

La méthode de [Wilk 87] s'applique également parfaitement aux sous-programmes conditionnels mais cette solution a été écartée pour éviter d'avoir trop d'informations à stocker en pile lors d'un sous-programme. En effet, dans une méthode DSM, la gestion du pointeur de références nécessite déjà des opérations de pile pour les sous-programmes. La méthode utilisée dans DJAM permet donc, par rapport à la solution de mise en pile de la signature, de diminuer la taille mémoire allouée à la pile et d'accélérer le traitement du moniteur lors des départs en sous-programme.

Il faut de plus remarquer que ce mode d'invariance des retours de sous-programmes peut être utilisé également pour des appels inconditionnels et qu'il s'applique parfaitement à une méthode ESM. Dans ce cas, il permet de s'affranchir complètement de la pile pour les sous-programmes, ce qui, pour une méthode ESM, est particulièrement appréciable et réduit considérablement le coût d'implantation[1].

La seule restriction concernant l'emploi de ce schéma d'invariance vient du fait qu'il faut être capable, au moment de la génération des informations de test, de faire la relation entre une instruction de retour de sous-programme et l'instruction correspondant au point d'entrée de ce sous-programme. En particulier, les points d'entrée multiples doivent être évités car il faudrait qu'ils aient la même signature intermédiaire, et la détermination de tous les points d'entrée pouvant accéder à une même instruction de retour n'est pas une chose aisée.

---

1 Dans une méthode DSM telle que DJAM, il existe déjà une pile pour les sous-programme et sa suppression n'est pas un argument.

Le fonctionnement de l'analyseur, pour les départs et retours en sous-programmes (condition vraie) doit assurer, en plus de l'invariance de la signature, la gestion du pointeur de référence et ceci est fait comme dans WDP, c'est-à-dire que le pointeur est empilé lors d'un départ en sous-programme, puis il est dépilé lors du retour. Dans la table des références, il n'y a donc pas de pointeur de prochaine référence associé à un retour de sous-programme, mais seulement une valeur d'ajustement.

Le fonctionnement de l'analyseur pour les départs et retours de sous-programmes est donc celui indiqué sur la figure 3.5 .

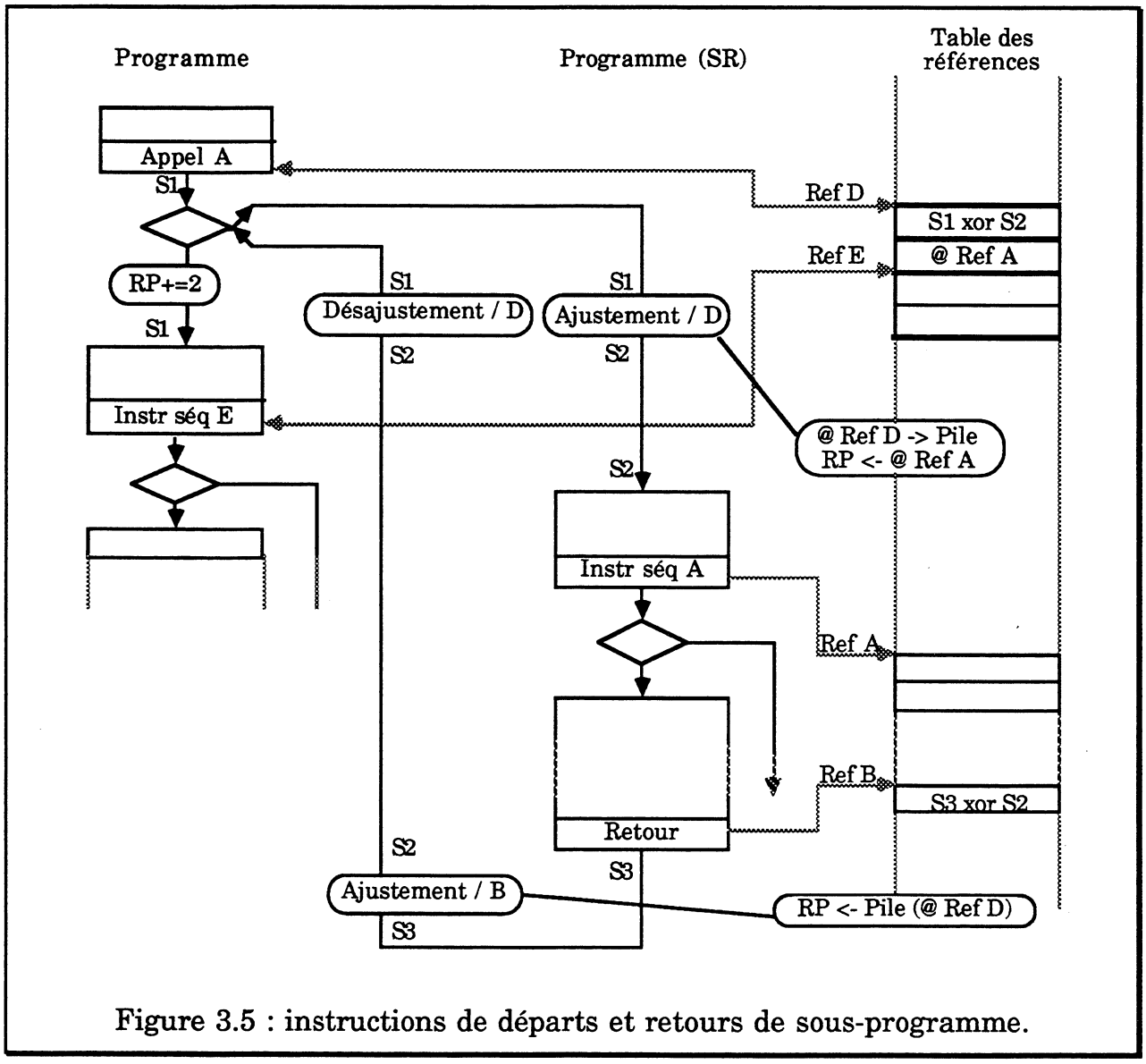


Figure 3.5 : instructions de départs et retours de sous-programme.

Lors d'un départ en sous-programme, en plus du schéma normal pour une rupture de séquence (ajustement plus mise à jour du pointeur), le pointeur de référence est empilé (avant mise à jour sur la référence destination).

Lors du retour de sous-programme, la valeur d'ajustement associée au retour est utilisée pour remettre la signature à sa valeur d'entrée de sous-programme. Après cet ajustement, le pointeur est dépilé et il retrouve la valeur qu'il avait avant l'appel du sous-programme : il pointe donc la référence associée à l'instruction de départ. Avec la valeur d'ajustement du départ en sous-programme, retrouvée grâce au pointeur dépilé, un désajustement est fait pour remettre la signature à la valeur qu'elle avait avant l'ajustement réalisé lors de l'appel du sous-programme. Pour finir, le pointeur de références est incrémenté de deux mots pour passer à la référence suivante.

3.2.4. Test de la signature

Pour le test de la signature, tous les arcs de séquencement étant déjà attribués aux ajustements, il faut nécessairement définir les instructions sur lesquelles doivent avoir lieu un test. A ces instructions correspond une valeur de test mais pas de pointeur de prochaine référence. En effet, la prochaine référence à accéder est la suivante dans la table des références car il n'y a pas de rupture de séquence provoquée par les instructions qui font un test, et le pointeur de références est simplement incrémenté lors d'un test.

Pour le repérage des instructions sur lesquelles a lieu un test, différentes solutions peuvent être envisagées :

- soit un code d'instruction particulier est choisi, et dans ce cas, le décodage des instructions indique l'emplacement et le type de ces singularités,
- soit un bit de mémoire étiquette marque ces singularités.

La première solution présente l'inconvénient du choix des instructions sur lesquelles doit avoir lieu un test. Un code d'instruction peu utilisé dans un programme, ou utilisé dans des circonstances particulières peut convenir. Un code définissable par programmation du moniteur est également envisageable, ce qui permet en fonction du programme vérifié d'obtenir une latence moyenne variable.

La seconde solution présente l'inconvénient de nécessiter une mémoire en parallèle de la mémoire instruction avec tous les problèmes que cela entraîne (coût matériel, difficulté de chargement).

Il reste également possible d'utiliser une technique de réduction de la latence de détection telle que celles décrites dans la section 1.4.4.2.5 , mais toutes deux utilisent également une mémoire en parallèle de la mémoire instruction (étiquette ou référence horizontale).

On considérera dans la suite de ce chapitre, pour les calculs de coûts mémoire, latence de détection et probabilité de masquage, que les tests de signature utilisent la première solution.

Le fonctionnement de l'analyseur pour le test de la signature est assuré suivant le principe de la figure 3.6 .

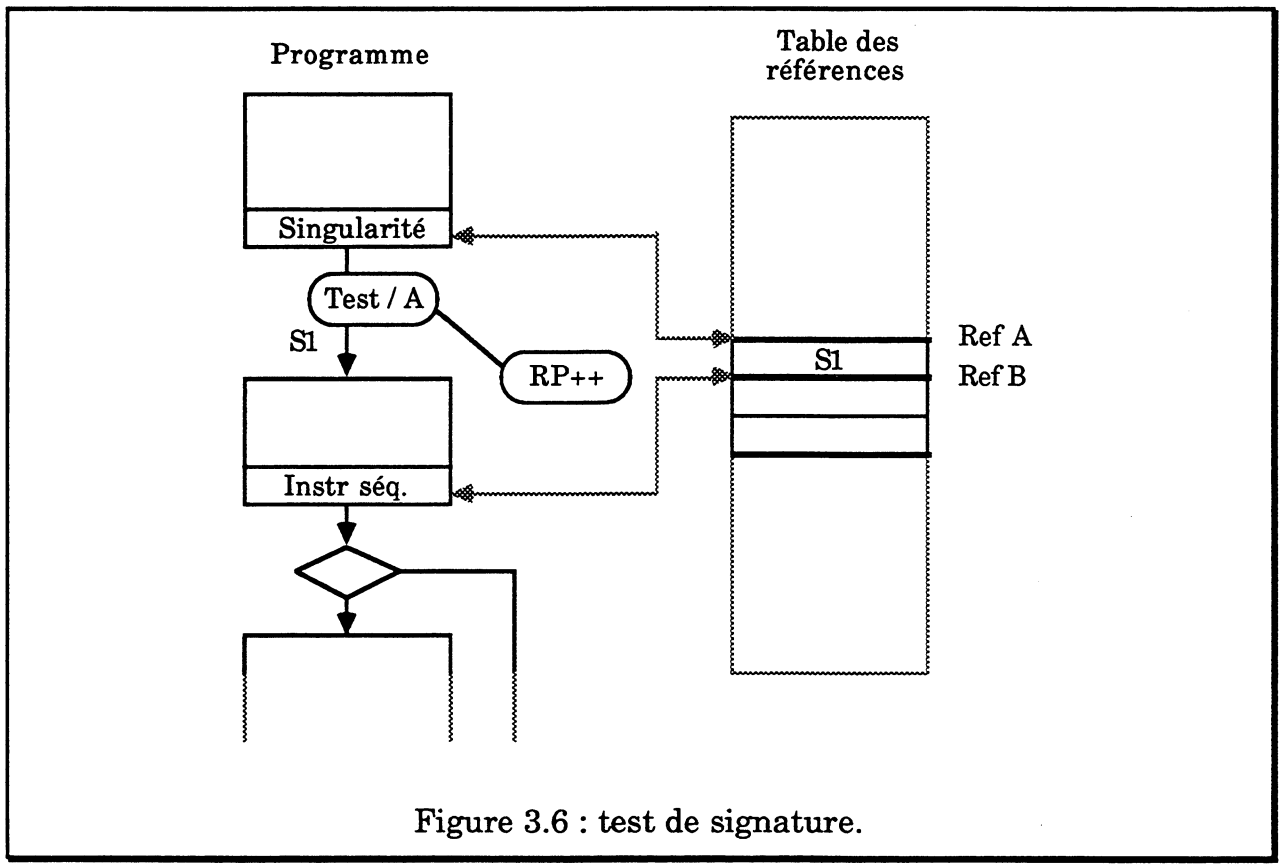


Figure 3.6 : test de signature.

3.2.5. Structure des informations dans la table des références

Trois types d'informations sont mémorisés dans la table des références :
- des signatures de référence,
- des valeurs d'ajustement,
- des données pour la gestion du pointeur sur les références.

Ces informations sont regroupées de trois manières différentes suivant le type de la singularité à laquelle elles sont associées, ce qui est représenté sur la figure 3.7 .

On remarquera donc, que contrairement à WDP, toutes les singularités n'entraînent pas le même nombre de mots dans la table des références. Ceci est gênant pour une lecture anticipée systématique des références car il n'existe pas d'informations, dans la table des références, sur le type de la singularité à venir. Ce type est en effet connu seulement au moment où l'instruction correspondante est lue

dans le programme vérifié ce qui permet alors de connaître le nombre de mots et la signification des données de référence pour traiter cette singularité.

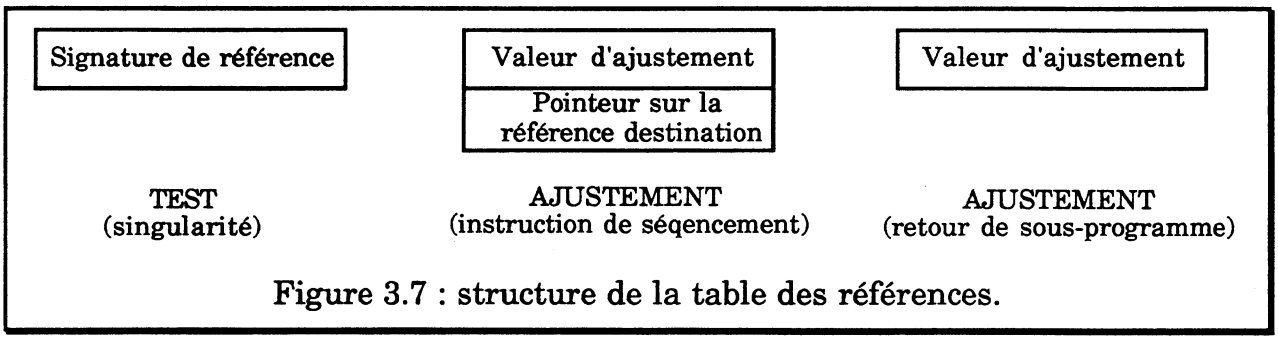


Figure 3.7 : structure de la table des références.

3.2.6. Fonctionnement du moniteur par type de singularité

Le fonctionnement du moniteur de DJAM est relativement simple par rapport au watchdog utilisé dans WDP. En particulier, le nombre de types de singularités est moins important ici car il n'est pas fait de distinction entre singularités conditionnelles et inconditionnelles. Les traitements ne sont donc effectués que sur occurrence d'une rupture de séquence (sauf pour les tests).

Dans cette section, les abréviations suivantes seront utilisées :

RP : pointeur de références géré par le moniteur,
SP : pointeur de la pile gérée par le moniteur,
SIGN : signature courante calculée par le moniteur,
Iμ : instruction du programme en cours d'exécution par le microprocesseur (μP).

3.2.6.1. Branchements

La structure des données correspondant à une instruction de branchement (conditionnelle ou non) est décrite sur la figure 3.8 :

- les données dans la table des références à l'adresse RPi correspondent aux valeurs utilisées par le moniteur pour traiter le branchement lorsque celui-ci a lieu ; s'il n'a pas lieu (instruction conditionnelle) alors ces données sont ignorées,
- les données dans la table des références à l'adresse RPj correspondent aux valeurs associées à la singularité située immédiatement après l'adresse Aμj dans le programme vérifié.

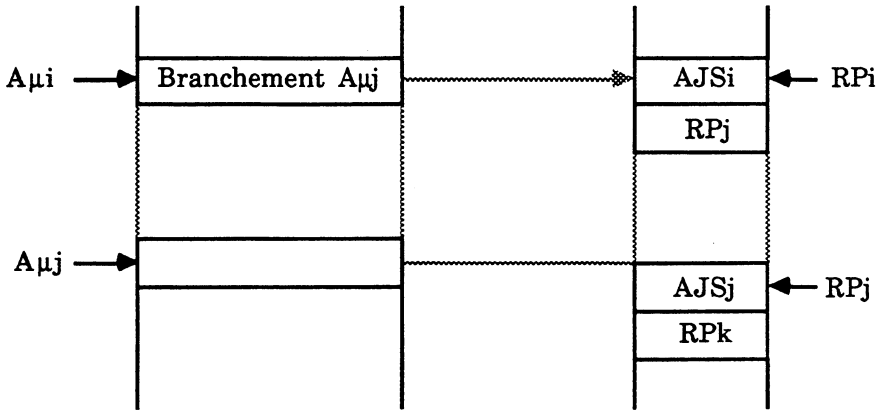


Figure 3.8 : instruction de branchement.

Fonctionnement du moniteur

quand Iμ = Branchement faire
si saut μP alors
SIGN ← SIGN ⊕ AJSi
RP ← RPj
sinon RP ← RPi + 2

3.2.6.2. Appels et Retours de sous-programme

La structure des données correspondant aux instructions d'appels et de retour de sous-programme (conditionnels ou non) est décrite sur la figure 3.9 :

- les données dans la table des références à l'adresse RPi correspondent aux valeurs utilisées par le moniteur pour traiter l'appel et le retour de sous-programme ; si l'appel (instruction conditionnelle) n'a pas lieu alors ces données sont ignorées,
- la donnée dans la table des références à l'adresse RPk correspond à la valeur utilisée par le moniteur pour traiter le retour de sous-programme ; si celui-ci n'a pas lieu (instruction conditionnelle) alors cette valeur est ignorée,
- les données dans la table des références à l'adresse RPj correspondent aux valeurs associées à la singularité située immédiatement après l'adresse Aμj dans le programme vérifié.

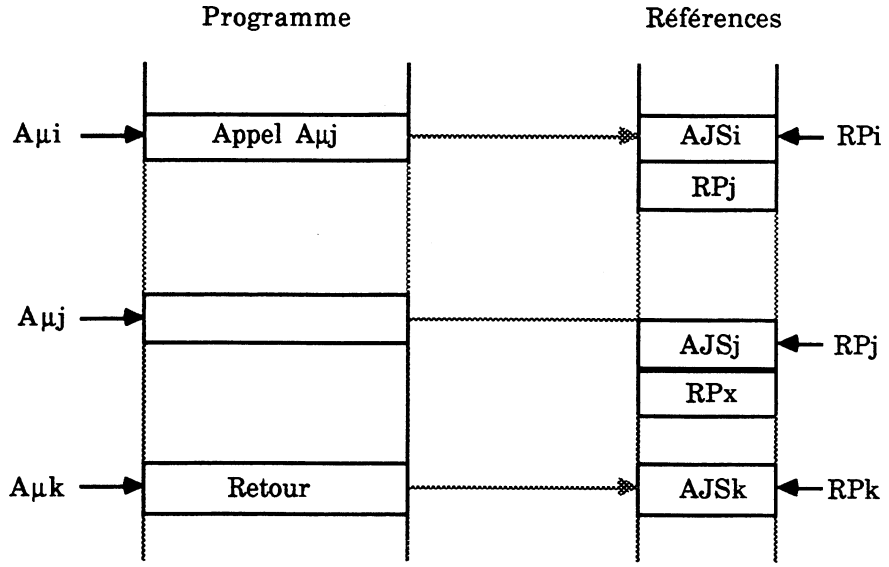


Figure 3.9 : instructions d'appel et retour de sous-programme.

Fonctionnement du moniteur à l'appel du sous-programme

quand Iμ = Appel de sous-programme faire
si saut μP alors
SIGN ← SIGN ⊕ AJSi
SP ← SP - 1
PILE(SP) ← RPi
RP ← RPj
sinon RP ← RPi + 2

Fonctionnement du moniteur au retour du sous-programme

quand Iμ = Retour de sous-programme faire
si saut μP alors
SIGN ← SIGN ⊕ AJSk
RP ← RPi = PILE(SP)
SP ← SP + 1
SIGN ← SIGN ⊕ AJSi
RP ← RPi + 2
sinon RP ← RPk + 1

3.2.6.3. Test de la signature

La structure des données correspondant aux singularités de test est décrite sur la figure 3.10 :

- la donnée dans la table des références à l'adresse RPi correspond à la signature de référence utilisée par le moniteur pour tester la signature au niveau de l'instruction située à l'adresse Aμi dans le programme vérifié,
- la référence suivante est située immédiatement après dans la table des références car il n'y a pas de rupture de séquence sur une singularité de test.

Fonctionnement du moniteur

quand Iμ = Singularité de test faire
si SIGN ≠ REFi alors ALARME
RP ← RPi + 1

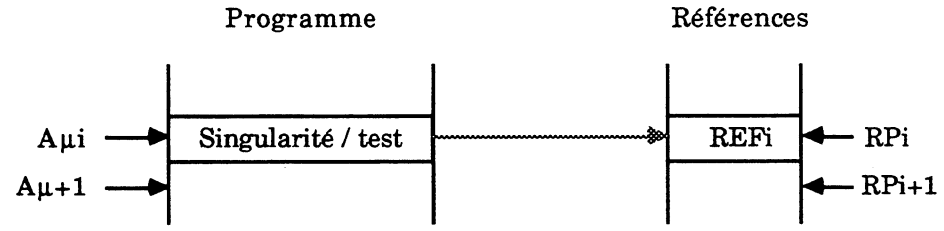


Figure 3.10 : singularité de test.

3.3. Génération des informations de test

3.3.1. Principe et limitations

La génération des informations de test se fait, comme pour toutes les méthodes DSM, directement sur le code exécutable du programme vérifié.

Etant donné que la méthode DJAM est une méthode avec ajustements, le programme de génération des références ne peut pas être aussi simple que pour WDP qui réinitialise la signature. L'algorithme est donc légèrement différent de celui de la méthode WDP. Il nécessite trois phases pour DJAM, alors que deux phases suffisent pour WDP. Par ailleurs, du fait de la suppression des données relatives aux points de jonction dans les informations de test de DJAM, il est nécessaire d'utiliser une structure de données temporaire avant de générer les informations de test définitives.

Cependant, par rapport à une méthode ESM avec ajustements, la présence dans DJAM d'un ajustement sur tous les arcs de séquencement supprime le

problème du placement des ajustements, mais bien entendu, le nombre d'ajustements est supérieur au nombre minimal théorique.

Comme pour WDP, on suppose qu'il n'existe pas de variables ou de constantes à l'intérieur d'une section contenant des codes d'instructions. Ceci permet d'affecter les signatures intermédiaires en tout point du programme en parcourant linéairement le code du programme et permet de s'affranchir de suivre les chemins maximaux du programme.

Pour l'affectation des signatures intermédiaires des destinations qui ne sont pas des points de jonction, une valeur arbitraire est choisie de telle sorte qu'elle soit différente du résultat de la compaction du bloc linéaire situé immédiatement avant cette instruction. Ceci permet de détecter les cas où la rupture de séquence de l'instruction de branchement (inconditionnel) précédant cette instruction n'a pas lieu.

Compte tenu du mode d'invariance de la signature des retours de sous-programme, on suppose de plus que tous les sous-programmes ne comportent qu'un seul point d'entrée et que ce point d'entrée précède, dans le code du programme, toutes les instructions de retour de ce sous-programme.

3.3.2. Algorithme de calcul

L'algorithme de calcul des références est décrit ci-dessous phase par phase :

- la première phase de l'algorithme consiste, comme dans WDP, à identifier les destinations des instructions de séquencement,
- la seconde phase de l'algorithme calcule la signature intermédiaire aux instructions destination et calcule les ajustements des retours de sous-programme,
- la troisième phase calcule les valeurs des ajustements associés aux instructions de séquencement (sauf retours de sous-programme) et génère la table définitive des informations de test.

L'algorithme de calcul des références de DJAM utilise une structure de données temporaire qui ressemble beaucoup au programme de vérification du watchdog de WDP. Les deux algorithmes de génération sont donc très voisins. Les étapes communes aux deux algorithmes sont la phase 1, le classement de la liste des noeuds et la recherche d'une destination dans la liste des noeuds. Hormis la fin de la phase 2, l'organigramme de l'algorithme de calcul des références de DJAM est donc identique à celui de WDP (Figure 2.5, cf 2.3.2) et c'est pourquoi la description est ici sous forme textuelle. De même, la lecture de l'algorithme de DJAM (plus détaillé) peut aider à comprendre la génération des références pour WDP.

Les structures suivantes sont utilisées:

```
{ IP        pointeur sur le programme }
{ RP        pointeur sur les références }

structure destination
            adresse,
            signature,
            RP,
fin structure;

liste_destination : liste(destination);
```

PHASE 1

La première phase de l'algorithme consiste, comme dans WDP, à identifier les destinations des instructions de séquencement :

```
IP = début_programme;

tant que IP <= fin_programme faire
            instruction = programme(IP);
            IP := IP + 1;
            si instruction = instruction_de_séquencement alors
                        calculer adresse_destination;
                        ajouter adresse_destination dans liste_destination;
                        si instruction = départ_sous_programme alors
                                    marquer destination dans liste_destination;
                        fin si;
            fin si
fin tant que; { phase 1 }

trier liste_destination par adresses croissantes;
```

PHASE 2

La seconde phase de l'algorithme calcule la signature intermédiaire aux instructions destination et calcule les ajustements des retours de sous-programme :

```
IP = début_programme;
RP = début_références;
prochaine_destination = début_liste_destination;
signature = entrée_programme;

tant que IP <= fin_programme faire
        tant que IP < prochaine_destination faire
                instruction = programme(IP);
                IP := IP + 1;
                compacter(instruction);
                si instruction = instruction_séquencement alors
                        RP = RP + 2;
                fin si;
                si instruction = séquencement_inconditionnel alors
                        nouvelle_signature;
                        { pas de chemin linéaire après }
                fin si;
                si instruction = retour_sous_programme alors
                        références(RP) = signature xor signature_SR;
                        RP = RP + 1;
                fin si;
        fin tant que;
        { IP = prochaine_destination }
        mémoriser signature dans liste_destination(prochaine_destination);
        mémoriser RP dans liste_destination(prochaine_destination);
        si prochaine_destination = entrée_sous_programme alors
                signature_SR = signature;
        fin si;
        prochaine_destination = adresse_suivante(liste_destination);
fin tant que { phase 2 }
```

PHASE 3

La troisième phase calcule les valeurs des ajustements associés aux instructions de séquencement (sauf retours de sous-programme) et génère la table définitive des informations de test :

```
IP = début_programme;
RP = début_références;
signature = entrée_programme;

tant que IP <= fin_programme faire
        instruction = programme(IP);
        IP := IP + 1;
        compacter(instruction);
        si instruction = instruction_séquencement alors
                calculer adresse_destination;
                rechercher adresse_destination dans liste_destinations;
                références(RP) = signature xor
                        signature(adresse_destination);
                références(RP+1) = RP(adresse_destination);
                RP = RP + 2;
        fin si;
        si instruction = séquencement_inconditionnel alors
                nouvelle_signature;
                { pas de chemin linéaire après }
        fin si;
fin tant que; { phase 3 et algorithme }
```

3.4. Capacité de détection d'erreur

Dans cette section, les erreurs de bits et les erreurs de séquencement seront distinguées. Les définitions sont les mêmes qu'à la section 2.4 :

- les erreurs de bits correspondent à une modification d'une ou plusieurs instructions sans altération du séquencement du programme,
- les erreurs de séquencement sont les ruptures de séquence inopinées et les destinations de branchements erronées ; ces erreurs sont considérées ici d'une manière identique.

3.4.1. Erreurs de bits

La probabilité de masquage de DJAM dans le cas d'erreurs de bits est exactement la même que pour les méthodes ESM avec ajustement et dépend uniquement de la probabilité de masquage de la compaction. Cette probabilité dépend du nombre d'erreurs qui sont introduites dans le dispositif de compaction. Il faut alors considérer plusieurs cas :

- la même erreur est compactée plusieurs fois (erreur permanente dans une boucle ne comportant pas de test),
- plusieurs erreurs sont compactées plusieurs fois (erreurs permanentes dans une boucle ne comportant pas de test),
- plusieurs erreurs sont compactées une seule fois (erreurs hors d'une boucle, dans une suite d'instructions ne comportant pas de test).

Dans le dernier cas, la probabilité d'une erreur de séquencement croit avec le nombre d'erreurs compactées, et en cas d'erreur n'affectant pas le séquencement, une probabilité de masquage asymptotique égale à 2^{-k} peut être retenue car le modèle d'indépendance des erreurs est vérifié (cf 1.2.2.2.2).

Dans les deux premiers cas, la probabilité de masquage dépend du nombre de fois où les erreurs sont compactées et, l'indépendance de ces erreurs n'étant pas vérifiée, une probabilité de masquage asymptotique égale à 2^{-k} ne peut pas être assurée. L'expérience montre cependant que le masquage reste faible [Delo 93].

Pour les erreurs de bits, l'efficacité de la méthode DJAM, et des méthodes avec ajustement en général, est donc inférieure à celle des méthodes qui réinitialisent la signature (WDP en particulier), car dans ces dernières, il n'existe jamais de boucles sans test :

- le cas où un seul mot est erroné dans une boucle, ou plus généralement dans un bloc linéaire, est ainsi détecté à 100% dans WDP (ainsi que dans PSAb) avec une signature de la même taille que les mots compactés et ceci quelle que soit la fonction de compaction utilisée,
- le cas où plusieurs mots sont erronés dans une boucle, ou plus généralement dans un bloc linéaire, est détecté avec une probabilité égale à 2^{-k} si la taille du bloc est suffisamment importante pour arriver à cette valeur asymptotique. Dans le cas contraire, la formule {P} (cf 1.2.2.2.2) doit être utilisée.

3.4.2. Erreurs de séquencement

Comme dans les méthodes ESM avec ajustements, les erreurs de séquencement ne peuvent être détectées qu'à travers la signature. Dans ces méthodes ESM, un masquage d'une erreur de séquencement se produit si la

signature intermédiaire au niveau de la destination du branchement (erroné ou intempestif) est égale au contenu du dispositif de compaction à ce moment. La probabilité de masquage dépend donc essentiellement de la répartition des signatures intermédiaires sur l'ensemble du programme.

Pour la méthode DJAM, le cas des erreurs de séquencement est légèrement différent. En effet, en supposant une erreur de séquencement dont la destination a une signature intermédiaire égale au contenu du dispositif de compaction, le masquage de cette erreur n'est pas irrémédiable, contrairement aux méthodes ESM, car ici les informations d'ajustement ne font pas partie du programme. Les ajustements successifs effectués par le moniteur de DJAM ont donc peu de chances de respecter l'invariance de la signature sur la partie de programme erroné exécuté, car le pointeur de référence ne pointe pas sur les valeurs d'ajustement correspondant aux instructions exécutées par le processeur.

Comme pour la méthode WDP, la probabilité que les chemins du processeur et du moniteur se rejoignent en générant la même signature est quasiment impossible à calculer et ne peut pas être supérieure à la probabilité de masquage du MISR. On peut cependant supposer qu'il existe plus de cas de tels masquages avec DJAM qu'avec WDP car ici la latence de détection peut être beaucoup plus élevée.

L'efficacité de la méthode DJAM pour les erreurs de séquencement est donc a priori meilleure que pour les méthodes ESM avec ajustements grâce au fait qu'il existe un mécanisme externe au programme pour l'invariance de la signature.

Par contre la méthode WDP reste supérieure sur ce point car les erreurs de séquencement sont détectées avec une latence très faible et une probabilité qui tend rapidement vers 100% lorsque la latence croit.

3.4.3. Conclusion

Pour la probabilité de masquage de DJAM, on retiendra la valeur 2^{-k}, faute de pouvoir calculer une valeur plus réaliste dans le cas particulier des erreurs permanentes dans les boucles du programme ainsi que dans les cas d'erreurs de séquencement où cette valeur constitue une borne supérieure.

Il faut toutefois remarquer que les erreurs permanentes dans les boucles peuvent être détectées d'une autre manière que par l'analyse de la signature des instructions exécutées. Des checksums, effectués périodiquement sur le programme, peuvent par exemple aider à résoudre ce problème.

Par rapport aux autres méthodes, l'efficacité de DJAM est meilleure que celle des méthodes ESM avec ajustements, pour les erreurs de séquencement, et elle est

identique pour les erreurs de bits. Dans les deux cas elle reste inférieure à celle de WDP.

3.5. Calcul du coût mémoire

Pour évaluer le coût mémoire de la méthode DJAM, un calcul similaire à celui présenté à la section 2.5 a été fait, fondé sur la méthode de calcul utilisée dans [Wilk 88] et [Wilk 90].

3.5.1. Coûts mémoire minimums

Les coûts mentionnés ici sont uniquement des coûts mémoire "verticaux" minimums. La latence de détection de DJAM est infinie avec ce coût minimal, comme pour d'autres méthodes ESM (Hsurf). Les tableaux 3.1 et 3.2 récapitulent les coûts associés aux différentes méthodes, incluant DJAM. Les chiffres indiqués entre parenthèse représentent le nombre de test par bloc linéaire.

Tableau 3.1 : matrice des coûts mémoire verticaux minimum par structure et par méthode (en mots).

	Méthodes ESM							Méthodes DSM			
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	WDP	Wopt	**DJAM**
if then	2(1)	1(0)	2(1)	1(0)	1(0)	2(0)	6(1)	5(0)	4(1)	4(1)	**2(0)**
if else	3(1)	1(0)	2(1)	1(0)	2(0)	2(0)	9(1)	8(0)	8(1)	6(1)	**4(0)**
switch(3)	6(1)	3(0)	2(0,5)	3(0)	5(0)	6(0)	18(1)	17(0)	16(0,5)	12(0,5)	**10(0)**
while/for	3(1)	2(1)	2(1)	1(0)	2(0)	2(0)	9(1)	8(1)	8(1)	6(1)	**4(0)**
do	2(1)	2(1)	2(1)	1(0)	1(0)	2(0)	6(1)	5(1)	4(1)	4(1)	**2(0)**
call	1(1)	1(1)	0(1)	1(0)	1(0)	2(0)	3(1)	3(1)	2(1)	2(1)	**2(0)**
return	1(1)	1(1)	1(1)	1(1)	1(1)	0(0)	3(1)	2(1)	2(1)	2(1)	**1(0)**

Tableau 3.2 : coûts mémoire verticaux minimum, par méthode (en %).

	Méthodes ESM							Méthodes DSM			
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	WDP	Wopt	**DJAM**
coût (%)	10-25	6-16	6-15	6-14	7-18	10-25	30-75	27-67	23-58	20-50	**14-35**

On pourra constater sur les tableaux 3.1 et 3.2 (ou sur la figure 3.11), que la méthode DJAM présente un coût mémoire inférieur à WDP et Wopt, ce qui est normal et prouve que les ajustements permettent de diminuer le coût mémoire minimum, même dans le cas d'une méthode DSM.

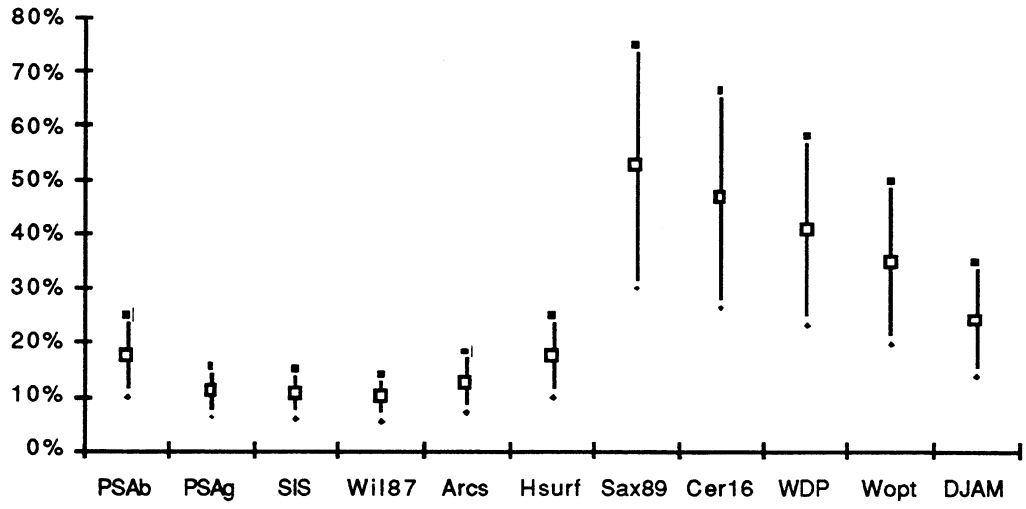


Figure 3.11 : coût mémoire vertical minimum de DJAM par rapport aux autres méthodes.

3.5.2. Coûts mémoire à latence égale

Les coûts mentionnés ici sont des coûts mémoire "verticaux" calculés en fonction de la latence de la méthode PSAb, c'est-à-dire un test par bloc linéaire. Les tableaux 3.3 et 3.4 récapitulent les coûts associés aux différentes méthodes, incluant DJAM.

Tableau 3.3 : matrice des coûts mémoire verticaux à latence égale par structure et par méthode (en mots).

	Méthodes ESM							Méthodes DSM			
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	WDP	Wopt	**DJAM**
if then	2	2	2	3	3	4	6	5	4	4	**3**
if else	3	3	2	5	6	6	9	8	8	6	**6**
switch(3)	6	6	4	9	11	12	18	17	22	18	**13**
while/for	3	2	2	5	6	6	9	8	8	6	**6**
do	2	2	2	3	3	6	6	5	4	4	**3**
call	1	1	0	3	3	4	3	3	2	2	**3**
return	1	1	1	1	1	2	3	2	2	2	**2**

Tableau 3.4 : coûts mémoire verticaux à latence égale par méthode (en %).

	Méthodes ESM							Méthodes DSM			
structure	PSAb	PSAg	SIS	Wil87	Arcs	Hsurf	Sax89	Cer16	WDP	Wopt	**DJAM**
coût (%)	10-25	9-23	6-16	19-46	20-50	24-60	30-75	27-67	24-60	21-52	**21-52**

On pourra constater sur les tableaux 3.3 et 3.4 (ou sur la figure 3.12), que la méthode DJAM présente un coût mémoire rigoureusement identique à celui de Wopt. Le coût de DJAM est par ailleurs inférieur à HSURF et il est très proche de la méthode "Arcs", toutes deux ESM avec ajustements. Ceci vient du fait que DJAM ne nécessite qu'un mot mémoire supplémentaire par test, alors que la plupart des méthodes ESM (PSAb et PSAg exceptées) nécessitent deux mots mémoire par test.

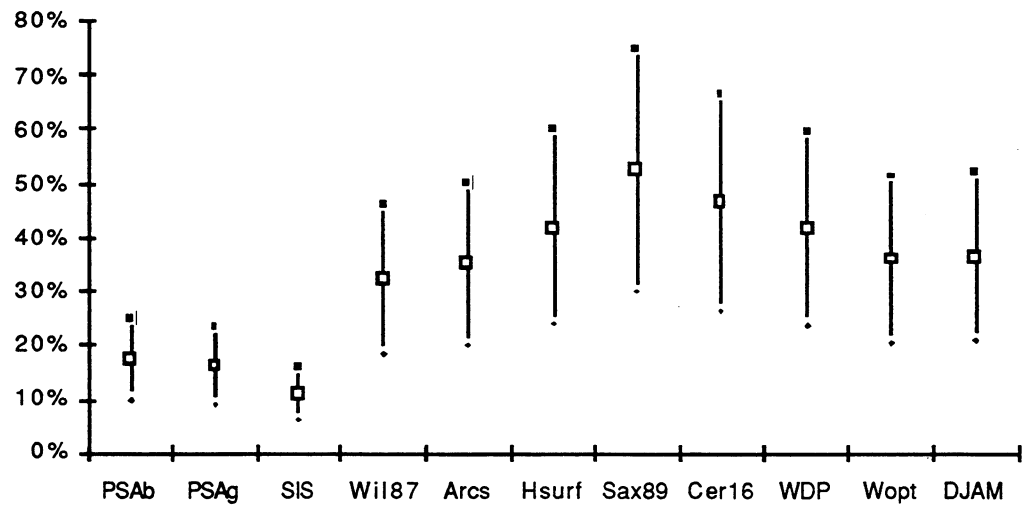


Figure 3.12 : coût mémoire vertical de DJAM, à latence égale, par rapport aux autres méthodes.

3.6. Prise en compte des caractéristiques du processeur

Pour la méthode DJAM, implantée d'une manière externe au processeur vérifié, seules les exceptions et la présence d'un pipeline sont à considérer parmi les caractéristiques particulières du processeur. La présence d'une MMU intégrée n'a aucune incidence étant donné que les adresses des singularités ne sont utilisées d'aucune manière. La présence d'une mémoire cache intégrée au processeur pour les instructions interdit l'implantation externe de la méthode, comme pour toutes les méthodes utilisant une signature dérivée.

3.6.1. Exceptions

Les exceptions peuvent être reconnues, comme dans WDP et sans surcoût matériel, par la détection d'une rupture de séquence intervenant ailleurs que sur une instruction de séquencement. La destination de cet arc de séquencement inopiné doit alors être, comme dans WDP, une instruction particulière permettant la mise à jour du pointeur de références sur l'adresse de la première référence de la routine d'exception.

La signature ainsi que le pointeur courant à l'instruction interrompue sont alors empilés avant le démarrage d'un nouveau processus de compaction.

3.6.2. Pipeline

La présence d'un pipeline pose un problème pour la génération de la signature en cas de rupture de séquence du processeur, car alors il existe des instructions lues mais qui ne sont pas exécutées. Ceci perturbe le fonctionnement d'un moniteur externe car la signature est générée à partir des instructions lues, et non pas exécutées. Ceci est vrai pour DJAM comme pour toutes les autres méthodes à signature dérivée.

Pour DJAM, il faut différencier le cas d'un pipeline synchrone et celui d'un pipeline asynchrone (cf 2.6.2.1). Dans les deux cas, la présence d'une file de signature est indispensable pour permettre l'empilement de la signature lors des départs en exceptions. La longueur de cette file de signature doit être égale au nombre de mots de la file d'instructions du processeur vérifiée (β).

3.6.2.1. Pipeline synchrone

La solution la plus simple consiste à inclure les codes des instructions lues mais non exécutées dans le calcul de la valeur d'ajustement [Delo 91].

3.6.2.2. Pipeline asynchrone

La solution précédente ne peut pas s'appliquer car le nombre de mots lus après l'instruction de branchement dépend des conditions d'exécution du programme et ne peut donc pas être connu au moment du calcul des valeurs d'ajustement.

La solution dans ce cas consiste à faire l'ajustement avec la signature intermédiaire de l'instruction de séquencement, comme s'il n'y avait pas de pipeline. Au moment d'une rupture de séquence, la signature dans la file, dont l'age est égal au nombre de mots lus depuis l'instruction de séquencement, sera utilisée pour faire l'ajustement.

Si plusieurs instructions de branchement se trouvent dans la file d'attente du processeur, à un niveau d'exécution potentielle, alors le moniteur de DJAM est

incapable d'assurer la cohérence de la signature car, contrairement à WDP, il n'est pas possible ici de connaître l'instruction responsable de l'arc de séquencement.

La méthode DJAM ne permet donc pas toujours la génération externe d'une signature quel que soit le pipeline des instructions du processeur. Seuls les processeurs disposant d'un pipeline synchrone sont entièrement supportés. Dans le cas d'un processeur avec pipeline asynchrone, il faut que le programme ne comporte pas de blocs linéaires de taille inférieure à la taille de la file des instructions du processeur. On considérera donc que la méthode DJAM ne s'applique pas à la vérification d'un processeur avec pipeline asynchrone, sauf cas exceptionnels.

3.7. Conclusion sur la méthode

La méthode DJAM est la première méthode DSM utilisant le principe des ajustements pour assurer l'invariance de la signature. Par ailleurs, dans cette méthode, la signature est toujours comparée avec une signature de référence unique, précalculée et mémorisée comme telle. Avec WDP, il s'agit des deux seules méthodes DSM procédant de cette sorte pour tester la signature.

Par rapport à la méthode WDP, dont elle est inspirée, la méthode DJAM présente un coût mémoire minimal inférieur pour le stockage des références ainsi qu'une plus grande simplicité de fonctionnement du moniteur. Par ailleurs, l'intégration du moniteur de DJAM à un processeur présente des avantages par rapport à un moniteur externe, alors que WDP est plus particulièrement conçue pour être implantée d'une manière externe.

L'efficacité de DJAM pour la détection d'erreurs est inférieure à celle de WDP. Cette efficacité reste par contre supérieure à celle d'une méthode ESM avec ajustements. Par ailleurs, la diminution de la latence d'erreur entraîne pour DJAM une croissance du coût mémoire inférieure à celle constatée sur la plupart des méthodes ESM.

Une implantation de DJAM sera présentée au chapitre suivant. Il s'agit d'une application à la vérification du flot de contrôle d'une unité centrale d'automate programmable.

Chapitre 4. Implantation de dispositifs d'analyse de signature dans une unité centrale d'automates programmables

L'étude menée sur des unités centrales (UC) d'automates programmables industriels a été faite en collaboration avec la société Télémécanique (Groupe Schneider). Etant donné que les détails techniques sur le fonctionnement d'une UC ont un caractère confidentiel, certaines descriptions sont volontairement peu approfondies. Par ailleurs, l'étude menée a été faite sur une base fonctionnelle dérivée des UC de la gamme d'automates TSX7 et non pas sur un produit commercialisé. Les descriptions techniques et résultats de mesures ne peuvent donc en aucun cas être rapprochés d'un produit existant.

Dans ce chapitre, après une description des objectifs de l'étude en section 4.1, l'architecture du prototype d'UC étudiée sera décrite en section 4.2 . Puis le principe des mécanismes de test ajoutés sur cette UC seront décrits en section 1.3 et leur implantation sera détaillée en section 4.4 . La section 4.5 est consacré à l'évaluation de l'efficacité des dispositifs de test de l'UC.

4.1. Objectifs de l'étude

Le but de l'étude menée sur les UC d'automates programmables est d'améliorer le niveau de sûreté d'un automate, en cas de défaillance interne de l'UC, par des moyens de test en ligne dont le coût de mise en œuvre reste faible.

Les erreurs ayant pour origine l'UC elle-même sont particulièrement importantes à détecter, car une partie non négligeable de la sûreté du contrôle d'un processus peut être assurée par l'UC elle même, à condition que les modes de défaillance du processus aient été pris en compte dans le développement de l'application [Mich 90]. Les modes de défaillance de l'UC, par contre, sont assez mal connus car une UC d'automate programmable actuelle est une machine complexe.

Par ailleurs, la détection d'erreurs de fonctionnement de l'UC par l'UC elle-même permet d'agir avant une éventuelle émission d'ordres erronés vers les modules d'entrées/sorties (E/S). Les efforts se sont donc concentrés plus particulièrement sur l'introduction de dispositifs de test en ligne dans l'UC elle-même.

Cependant, pour être dignes d'intérêt, ces dispositifs doivent présenter un coût d'implantation et de mise en œuvre assez faible pour que les caractéristiques générales de l'UC ne soient pas sensiblement affectées. Les performances, la capacité de traitement, l'encombrement, la consommation et le taux de défaillance,

doivent être préservés, dans la mesure du possible, ainsi que la compatibilité des programmes utilisateurs, ceci afin d'offrir une transparence quasi totale pour l'utilisation d'une UC disposant de moyens de test en ligne supplémentaires.

4.2. Architecture générale d'une UC

4.2.1. Architecture matérielle

La figure 4.1 présente l'architecture matérielle générale de l'UC étudiée. Celle-ci est organisée autour d'un microprocesseur Intel 80x86 (8086 ou 80386) et d'un ensemble de composants spécifiques (ASICs) dans lesquels se trouvent :

- un processeur spécifique pour les opérations booléennes (appelé par la suite BP pour Boolean Processor), qui exécute le code de l'application et qui est contenu physiquement dans un composant ASIC nommé "PIU",
- un dispositif de gestion du bus d'entrées/sorties (appelé par la suite "picoséquenceur"), contenu physiquement dans le composant PIU,
- de la logique de bus permettant les accès mémoire des différents processeurs, contenue physiquement en partie dans le PIU et dans une série de composants ASICs particuliers au modèle de 80x86 utilisé.

Différents composants mémoires sont utilisés sur l'UC pour stocker le code système, les données système et les données de l'application.

L'horloge temps réel (HTR) est assurée par un microprocesseur monochip Intel 8052 qui sert également à la gestion de la liaison console et à la surveillance partielle du fonctionnement de l'UC.

L'application exécutée par l'UC est stockée soit en RAM soit en PROM. Lorsqu'elle est en PROM, l'application est physiquement contenue dans une cartouche PROM. Lorsqu'elle est en RAM, par contre, l'application est contenue physiquement soit dans la RAM interne de l'UC, soit dans une cartouche RAM, soit dans les deux à la fois.

Une application est constituée de plusieurs segments (au sens 8086) dont certains contiennent le code de l'application exécutable par le processeur spécifique BP (Boolean Processor).

Le code système 80x86 est contenu dans des PROMs mais, pour améliorer les performances, il peut également être installé dans une mémoire RAM rapide à accès direct qui se substitue aux PROMs sur une partie ou sur la totalité du système.

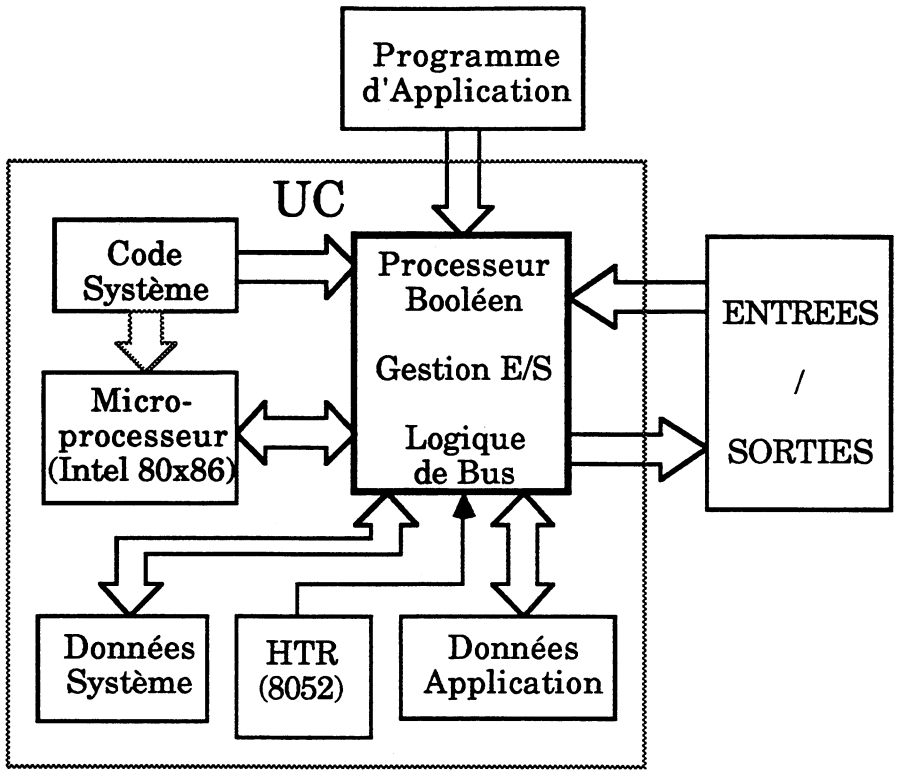


Figure 4.1 : architecture générale de l'UC étudiée.

4.2.2. Exécution de l'application

Une application exécutée par une UC se décompose en plusieurs tâches et le modèle de programmation utilisé ici propose sept tâches se répartissant en quatre niveaux de priorités. La tâche de priorité maximale est activée sur événement externe (interruption) et elle est appelée tâche IT. Toutes les autres tâches sont synchrones et sont activées par l'horloge temps réel du système. La période d'activation est définie par l'utilisateur. Ces tâches sont appelées :

- tâche FAST : tâche rapide, utilisée pour les signaux d'entrées/sorties dont la prise en compte est critique d'un point de vue temporel,
- tâche MAST : tâche principale de l'application,
- tâches AUX0 à AUX3 : tâches auxiliaires, utilisées pour décharger la tâche MAST ou pour la gestion d'entrées/sorties dont le temps de prise en compte n'est pas critique.

Lorsque plusieurs tâches sont activées en même temps, une tâche peut être interrompue par les tâches de niveaux de priorité supérieurs mais pas par les tâches ayant un niveau de priorité égal ou inférieur.

4.2.2.1. Cycle d'une tâche

Chaque tâche synchrone peut être individuellement validée ou arrêtée sur ordre explicite. Lorsqu'une tâche est activée, après une étape d'initialisation, elle suit un cycle de fonctionnement (Figure 4.2) divisé en au moins trois étapes fondamentales :

- échantillonnage des modules d'entrées attribués à la tâche (IMP_IN),
- exécution du code application programmé pour cette tâche (U_CODE),
- mise à jour des modules de sorties attribué à la tâche (IMP_OUT).

Les états IMP_IN et IMP_OUT correspondent à un processus d'entrées/sorties "implicite" sur les modules d'entrées/sorties assignés à la tâche. Des échanges d'entrées/sorties peuvent également avoir lieu, sur tout module d'entrées/sorties, pendant l'exécution du code application d'une tâche (U_CODE), mais ils doivent alors être programmés dans l'application par des instructions spécifiques. Ces échanges seront alors appelés "explicites".

En fin de cycle, après mise à jour des sorties, la tâche se trouve dans un état d'attente (INTER_SCAN) dont elle sort sur activation par le système temps réel. Ce cycle de fonctionnement se répète jusqu'à un ordre spécifique d'arrêt de la tâche ou de l'automate.

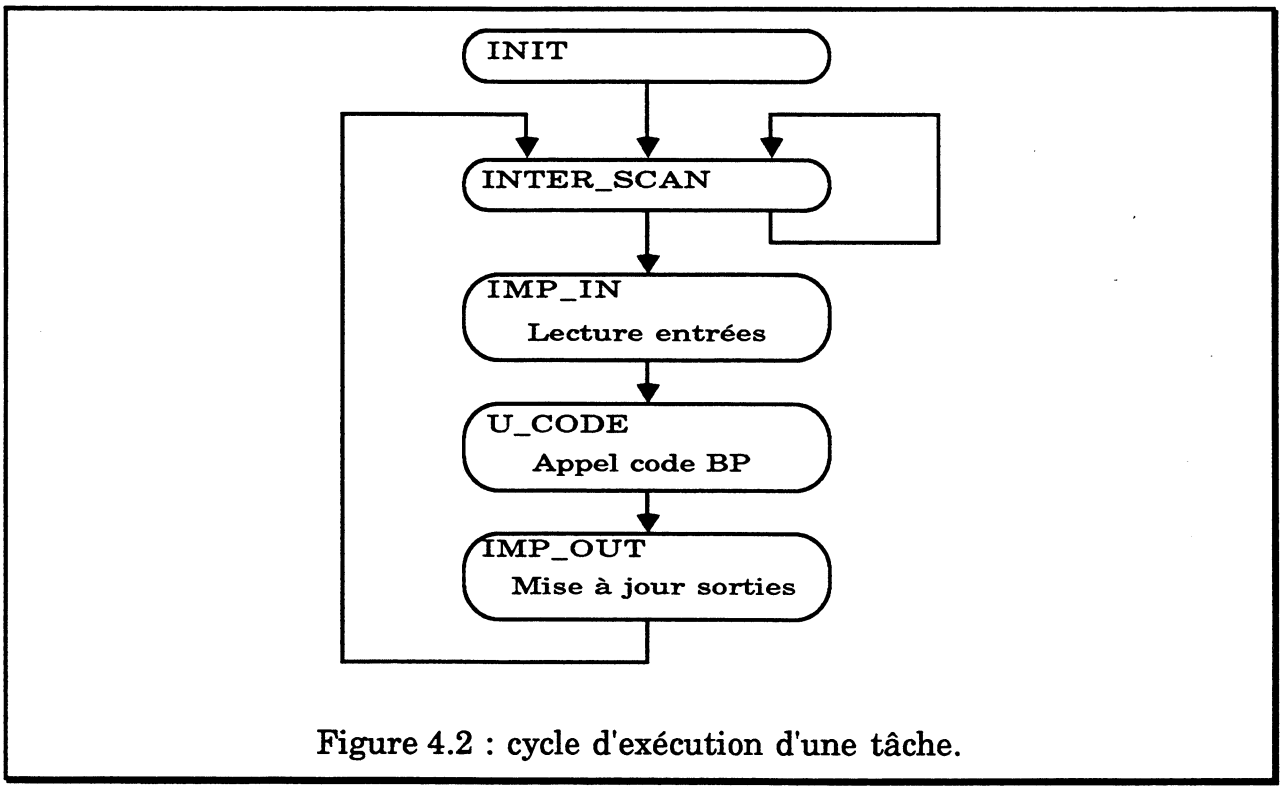


Figure 4.2 : cycle d'exécution d'une tâche.

4.2.2.2. Exécution du code application d'une tâche

Le programme application exécuté à chaque cycle d'une tâche est programmé dans un langage propre aux automates Télémécanique (PL7_3). Ce programme est ensuite compilé dans un code exécutable par le processeur BP.

Lorsqu'une tâche se trouve dans l'état d'exécution du code application (U_CODE), le processeur BP est lancé par le système. Cependant, comme le processeur BP n'est capable de traiter que des données de type "bit" (d'où son nom), tous les traitements de données autres que booléennes sont en fait exécutés par le microprocesseur 80x86 (appelé NP pour "Numerical Processor").

Les processeurs BP et NP réunis forment ainsi une machine capable d'exécuter, sous contrôle du système, l'application programmée. D'un point de vue matériel, les deux processeurs ont un fonctionnement exclusif l'un de l'autre : pendant que l'un des deux processeurs travaille, l'autre attend que le processeur en action lui redonne la main. Les différents niveaux de machine d'une UC sont représentés sur la figure 4.3 .

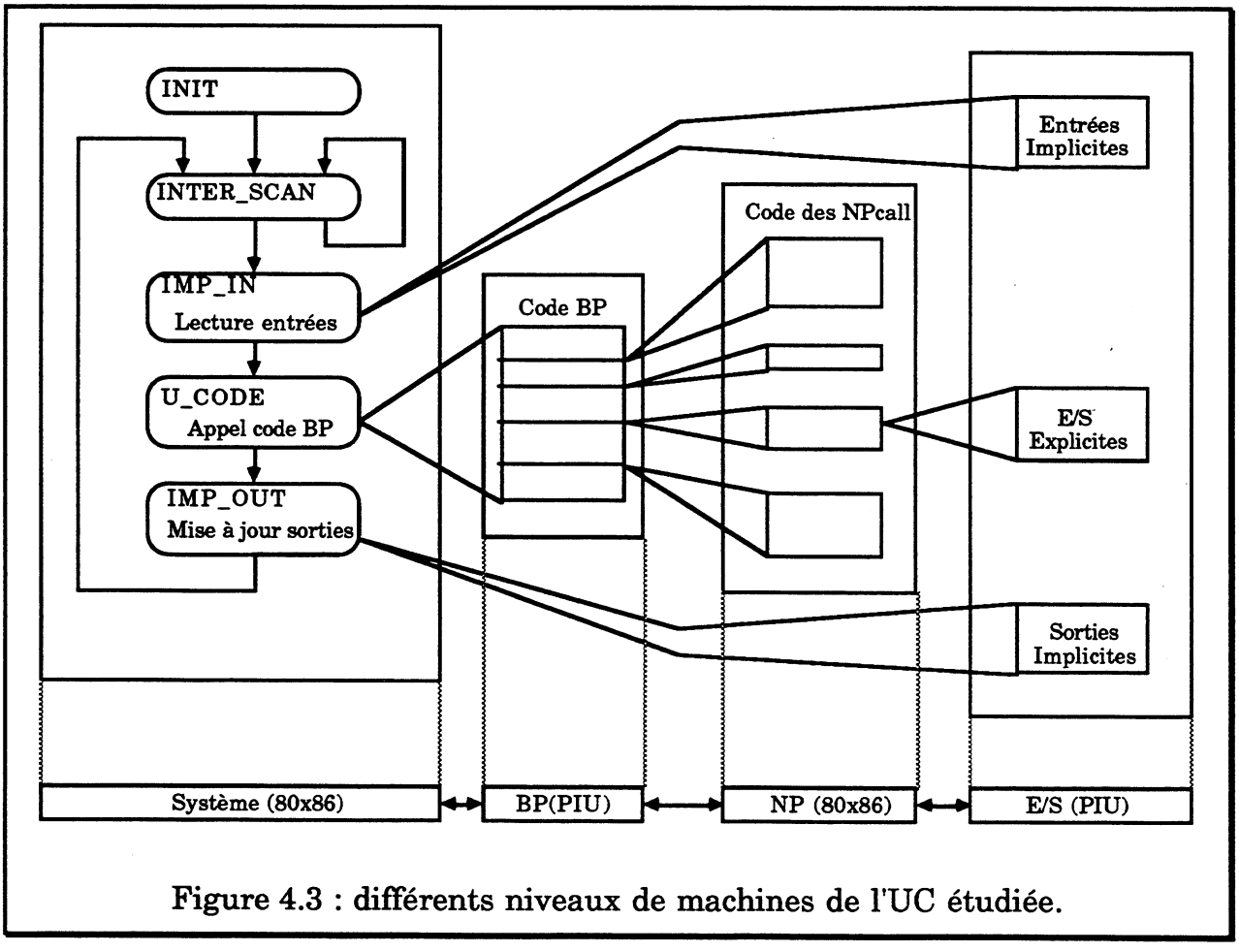


Figure 4.3 : différents niveaux de machines de l'UC étudiée.

Dans le code exécutable par BP, certaines instructions sont donc en fait des appels au système, pour l'exécution de séquences de code 80x86 particulières. Le processeur 80x86 exécutant ces parties de code est alors appelé NP (pour "Numerical Processor", par opposition à "Boolean Processor") et les instructions du code BP dont l'effet est un appel au système sont donc appelées très logiquement des "NPcall". On peut donc considérer que le processeur BP exécutant l'application est un processeur spécifique dont le jeu d'instruction peut être étendu par émulation logicielle d'instructions.

Sur l'architecture étudiée, parmi les NPcall, on trouve entre autres toutes les instructions de séquencement du processeur BP. Le type de donnée traité par ces NPcall de séquencement est en fait simplement le pointeur BP sur le code application. Ce pointeur est contenu physiquement dans le BP (PIU) qui l'incrémente au fur et à mesure de la lecture du code application, les ruptures de séquences étant sous-traitées par des NPcall. Ce pointeur sera par la suite appelé "BPIP" (Boolean Processor Instruction Pointer).

4.2.3. Dispositifs de test en ligne de base

Les dispositifs décrits dans cette section sont des mécanismes de test en ligne tout à fait classiques compte tenu de l'architecture étudiée. Ils sont présents sur certains modèles de la gamme d'UC TSX 7 .

4.2.3.1. Moyens matériels

a/ Parité en lecture mémoire.

Une mémoire de parité est présente pour la vérification de la mémoire utilisée par l'UC, à l'exception de celle contenant le code système (en PROM). L'espace mémoire contrôlé comprend donc la RAM interne et la cartouche. Deux types d'erreurs peuvent être détectés par la parité :

-> erreur de valeurs (défaut mémoire par exemple),

-> erreur d'adressage (tentative d'accès à une zone inutilisée).

Pour les erreurs de valeurs, la parité détecte toutes les erreurs n'affectant qu'un nombre impair de bits soit 50% de l'ensemble des erreurs pouvant affecter un mot mémoire et 100% des erreurs n'affectant qu'un seul bit dans un mot mémoire.

Des erreurs d'adressage peuvent être également détectées par la parité en cas de tentative d'accès mémoire dans une zone où la parité n'est pas initialisée. Il s'agit en fait d'une forme de "capability checking" [Namj 82b].

Des erreurs d'adressage peuvent avoir pour origine :

- un défaut dans la logique de bus,

- la sortie d'un pointeur programme de la zone contenant du code (système ou application),
- une erreur dans un calcul de pointeur permettant d'accéder une structure de donnée en mémoire (système).

La parité étant toujours initialisée seulement sur l'espace occupé physiquement par la cartouche, le pouvoir de détection de la parité est donc d'autant meilleur que la taille de la cartouche est petite. En cas de cartouche de taille inférieure à l'espace mémoire maximal contrôlé, la probabilité de détection d'une erreur d'adressage par la parité est de 0,5 par accès mémoire dans une zone inoccupée (donc non initialisée). En cas de défaut bus permanent ou en cas de sortie d'un pointeur programme de sa zone de code, la probabilité de détection tend rapidement vers 1 car pour N accès mémoire dans une zone non initialisée, cette probabilité de détection est égale à :

$$P_{detect} = \sum_{i=1}^{N} \frac{1}{2^i}$$

b/ Chien de garde temporel (TO52).

Un chien de garde temporel, implanté sur toutes les UC, permet de détecter les types d'erreurs suivant :

-> défaut matériel bloquant (processeur ou logique de bus),

-> boucle infinie du microprocesseur 80x86.

Ce chien de garde temporel est rafraîchi périodiquement par le microprocesseur de surveillance 8052, en fonction de l'observation de l'activité du 80x86.

4.2.3.2. Moyens logiciels

a/ Débordement temporel d'une tâche (overrun).

Un défaut logiciel bloquant est détecté lorsqu'une tâche déborde de son temps de cycle pendant plus de huit cycles d'activation consécutifs. Ce type de défaut peut être provoqué par :

- une surcharge anormale de l'UC,
- une boucle infinie dans l'exécution d'une tâche.

b/ Code opératoire BP illégal.

Les codes opératoires invalides ne sont pas, pour la plupart, détectés par le processeur BP et l'exécution d'un code non défini se traduit par un NOP (No OPération). Cependant, tous les codes de NPcall ne sont pas attribués dans le jeu d'instruction du processeur BP et l'exécution par le BP d'un NPcall dont le code

n'est pas utilisé provoque l'exécution par le 80x86 d'un "NPcall erreur" qui stoppe l'UC en défaut mémoire.

c/ Défauts d'E/S.

Deux types de défauts de l'UC peuvent se traduire par un défaut d'E/S :
- erreur de protocole pendant un échange d'E/S,
- problème dans la configuration (vérification périodique).

L'UC n'est pas stoppée d'elle-même en cas d'erreur d'E/S : la détection d'une telle erreur se traduit par l'allumage de la diode I/O en face avant de l'UC et par le positionnement d'indicateurs dans les bits systèmes de l'automate. Le contrôle de ces indicateurs est laissé à la responsabilité de l'application.

d/ Checksum périodiques.

Des checksums sont effectués périodiquement pour vérifier l'intégrité du code système (80x86) sur les parties de code contenues en RAM.

e/ Auto-tests au démarrage.

Des auto-tests sont exécutés sur redémarrage à froid de l'UC. Ils ne seront pas détaillés étant donné que l'étude ne porte que sur les possibilités de test en ligne des UC.

4.2.3.3. Exceptions du microprocesseur 80386

Sur le prototype d'UC étudié, un microprocesseur Intel 80386(sx) est utilisé en mode réel. La compatibilité logicielle avec un 8086 est totale d'un point de vue fonctionnel mais par contre, en cas d'erreur, les deux microprocesseurs ne sont plus tout à fait équivalents. Un 80386 dispose des modes de détection d'erreur suivants :

-> débordement du segment de pile,
-> codes opératoires illégaux,
-> instructions privilégiées,
-> débordement d'adressage.

Seul le premier type d'erreur est détecté par un 8086. La détection des trois derniers types d'erreurs sert à vérifier que seul est utilisé le sous-ensemble du 80386 équivalent à un 8086. En particulier, la capacité d'adressage (logique et physique) d'un 8086 est de 20 bits, alors que la capacité d'adressage logique d'un 80386 (sx ou dx) est de 32 bits. En cas de calcul d'adresse se traduisant par une adresse de taille supérieure à 20 bits, le 8086 effectue un rebouclage cyclique dans son espace d'adressage, alors qu'un 80386 part en exception pour violation des frontières d'adressage 8086.

4.3. Vérification du flot de contrôle d'une UC

L'idée de vérifier le flot de contrôle d'une UC parait assez naturelle compte tenu du fait qu'une UC d'automate programmable fonctionne comme la partie contrôle de l'automate en envoyant des ordres au processus commandé. Par ailleurs, le faible coût de ce type de méthode de test en ligne correspond tout à fait aux critères requis.

Le GFC de l'UC correspond à celui du programme d'application et, étant donné le fonctionnement d'une UC, le niveau de contrôle le plus élevé, pendant l'exécution du code application, est celui du processeur BP. En effet, pendant les phases d'exécution du code application, le microprocesseur 80x86 est vu comme une machine esclave, alors que le BP est le maître. Une vérification de flot de contrôle à deux niveaux peut donc être faite sur une UC en vérifiant le flot de contrôle de chaque processeur.

4.3.1. Principe du test

Pour l'implantation des dispositifs nécessaires à la vérification du flot de contrôle des deux processeurs de l'UC, l'architecture des UCs a été utilisée au mieux afin de limiter les coûts d'implantation. En particulier, le principe des NPcall et le fait que le processeur BP soit un processeur spécifique, intégré sous forme d'ASIC, ont été mis à profit pour limiter au maximum le coût matériel des dispositifs implantés. De plus, du fait du fonctionnement exclusif des deux processeurs, le dispositif de génération de signature a été mis en commun. Ceci permet de limiter le coût matériel des dispositifs mais l'intérêt essentiel est de pouvoir combiner la signature des deux processeurs en une seule et unique signature représentative des appels successifs aux différents niveaux de machines (cf figure 4.3, section 4.2.2.2).

Cependant, pour que la vérification du flot de contrôle des deux processeurs soit réellement efficace, il faut de plus qu'aucune erreur détectable par un de ces mécanismes ne puisse être propagée sur les sorties de l'automate avant sa détection. Pour cela, un dispositif matériel a été connecté au processeur d'entrées/sorties (PIU) pour inhiber les échanges d'entrées/sorties tant qu'un test de signature positif n'a pas eu lieu. Ce dispositif permet d'assurer que le système et l'application sont dans un état correct avant tout échange d'entrées/sorties et empêche donc toute mise à jour intempestive des sorties, phénomène dont les conséquences au niveau du processus contrôlé sont aléatoires.

4.3.2. Analyse de signature sur le code application

4.3.2.1. Génération de la signature

Pour le processeur BP, il a été implanté un dispositif d'analyse de signature permettant une génération de signature dérivée sur le code de l'application. Le dispositif de génération de signature a pu être placé directement dans le processeur BP (ASIC PIU) avec tous les avantages que cela implique :

- génération de signature à partir des instructions exécutées et non pas seulement lues (pas de problèmes avec la file d'instructions),
- possibilité de compacter différemment les codes opératoires et les paramètres,
- décodage des instructions déjà effectué par le processeur,
- pas de problème pour la reconnaissance des interruptions.

Pour assurer une meilleure détection, les paramètres des instructions sont compactés en inversant certains bits de la donnée lue. Une erreur se traduisant par le non-respect du format d'une instruction sera ainsi détectée par l'analyse de signature. Ce principe est exactement le même que celui utilisé dans HSRUF32 (cf figure 1.21, section 1.6.2.1). Dans le cas du processeur booléen, les paramètres des NPcalls, entre autres, sont compactés différemment des instructions.

4.3.2.2. Implantation de la méthode DJAM

Pour le stockage des informations de test, une méthode DSM a été utilisée car une contrainte majeure était de respecter la compatibilité du code BP. Cette compatibilité logicielle est nécessaire pour qu'une application générée sur une UC ne disposant pas d'analyse de signature puisse être exécutée sur une UC avec analyse de signature en bénéficiant de ce mécanisme de test, et ceci sans avoir à faire une recompilation.

Par ailleurs, l'utilisation d'une méthode DSM permettait de ne pas avoir à modifier le compilateur du processeur BP, ce qui semblait préférable bien que ce compilateur ait été développé par Télémécanique.

L'utilisation d'une méthode DSM pour le processeur BP a donc été déterminée par l'obligation de respecter la compatibilité logicielle au niveau application. La méthode DJAM, présentée au chapitre 3 , a été retenue. La méthode WDP n'était en effet pas particulièrement bien adaptée au processeur BP pour plusieurs raisons :

- WDP est particulièrement bien adaptée pour une implantation séparée du processeur alors que pour le processeur BP, il s'agit d'une implantation interne (ASIC PIU),

- la latence de détection n'est pas critique pour le processeur BP dans la mesure où il existe des dispositifs complémentaires tels que l'overrun d'une tâche, le chien de garde temporel de l'UC et le dispositif rajouté de blocage des entrées/sorties,
- l'utilisation de WDP pour le processeur BP, bien qu'intéressante d'un point de vue efficacité, aurait conduit à un coût d'implantation inutilement élevé.

Etant donné que la latence de détection n'est pas critique, une méthode avec ajustements semble préférable à cause de la réduction du nombre de singularités dans le programme (diminution du coût mémoire).

Comme le moniteur est intégré au processeur, un repérage des singularités par adresses est inutilement coûteux dans la mesure ou le décodage des instructions est déjà fait par le processeur lui-même.

Pour une méthode DSM, le placement des ajustements sur arcs semble donc tout à fait indiqué et comme les instructions de séquencement du processeur BP sont exécutées par des NPcalls, l'accès aux valeurs d'ajustement peut être pris en charge par le 80x86 et non par un matériel spécifique.

Ceci réduit donc la complexité du moniteur sur plusieurs points :
- le décodage des instructions de séquencement est fait par le BP lui-même,
- la détection des ruptures de séquence n'est pas nécessaire car c'est l'exécution de l'instruction provoquant une rupture de séquence qui déclenche l'ajustement,
- l'accès aux références et la gestion du pointeur de références sont assurés par le 80x86, pendant l'exécution de certains NPcall, et ne nécessite donc pas de matériel particulier.

4.3.2.3. Test de la signature

Au chapitre 3, on a vu que le test de la signature avec la méthode DJAM devait être fait sur un type d'instruction particulier. Dans le cas du code BP, il existe une instruction dont l'effet est de rendre la main au système en fin d'exécution du code application d'une tâche (NPcall ne rendant pas la main au BP mais au système). Ce type d'instruction a donc logiquement été choisi pour tester la signature lors du retour du contrôle au système. Le NPcall marquant la fin du code étant identique au NPcall qui marque le début du code, un test est également fait en début de code.

Un test de la signature a lieu également à chaque NPcall d'entrées/sorties explicites afin que le dispositif de blocage des E/S ne se déclenche pas. Une signature de référence est donc associée aux NPcalls E/S explicites et NPcalls début/fin de code. Mis à part le cas des NPcalls d'E/S explicites, il n'y a donc pas de test de

signature pendant l'exécution du code BP mais seulement au début et à la fin du code. La signature est donc testée au minimum deux fois lors de l'exécution du code application. Ce code étant exécuté à chaque cycle de la tache, la latence de détection est donc toujours inférieure au temps de cycle d'une tache.

Les points de test ainsi placés, hormis le test en début de code application, permettent d'avoir un test de signature uniquement avant chaque phase de mise à jour des sorties. Ceci minimise donc le coût mémoire associé aux références mais sans nuire à l'efficacité de la détection. En effet, le placement des points de test garanti qu'aucune erreur détectable par l'analyse de signature ne peut être propagées sur les sorties avant détection. Le principe du placement des points de test utilisé ici est tout à fait similaire à celui présenté dans [Mart 90].

Il faut cependant remarquer à ce sujet qu'un phénomène de masquage peut éventuellement se produire en cas d'erreurs mémoire permanentes dans une boucle du programme d'application (cf 3.4.1).

4.3.2.4. Génération des informations de test

Pour le calcul des références, l'algorithme décrit en section 3.3.2 est utilisé, toutes les contraintes liées à l'utilisation de celui-ci étant respectées par le compilateur du code BP. Sur le prototype réalisé, la génération des références peut être faite de deux manières différentes, soit sur la console de l'automate soit par l'UC elle même.

Dans les deux cas, les références sont contenues physiquement dans une extension optionnelle du système (OFB : Optional Function Bloc) dont le code est exécuté exclusivement dans l'état "INIT" de la tâche MAST et dont le rôle est d'initialiser les dispositifs de test.

Le calcul des références par la console de l'automate (compatible PC par exemple) est effectué à partir du fichier binaire contenant le code exécutable de l'application. Les références sont alors chargées sur l'automate en même temps que l'application. L'OFB de gestion des références est alors constitué essentiellement d'un segment de constantes contenant les références.

Le calcul des références par l'UC elle-même est effectué à partir du code de l'application résidente sur l'automate, pendant la phase d'initialisation de l'automate. L'OFB de gestion des références est alors constitué d'un segment de code plus volumineux, contenant le programme de calcul des références, et d'un segment de données dans lequel les références sont stockées après calcul. Le calcul des références est exécuté une fois pour toutes lors du démarrage de l'exécution du programme d'application

L'avantage d'un calcul des références par l'UC est la transparence totale au niveau de l'utilisation de l'analyse de signature. Par contre, en cas de modification indésirable du programme utilisateur (erreur d'un opérateur ou malveillance), les références étant recalculées au démarrage de l'automate, l'analyse de signature ne sera d'aucun recours.

A l'inverse, lorsque le calcul des références est réalisé sur la console, toute modification du programme, même volontaire, sera détectée par l'analyse de signature si les références ne sont pas explicitement mises à jour. Ce mode de génération des références est donc bénéfique sur le plan de la sécurité informatique.

4.3.3. Test au niveau système

La présence d'une vérification du flot de contrôle du processeur BP permet dans une certaine mesure de tester le fonctionnement du processeur 80x86. En effet, une erreur de fonctionnement pendant l'exécution d'un NPcall de séquencement se traduira par un séquencement erroné du processeur BP et ceci sera détecté a priori dans tous les cas, soit par l'analyse de signature du code BP si la destination du déséquencement est une zone de code BP, soit par les moyens de test de base de l'UC (overrun d'une tâche, NPcall erreur).

De même un déséquencement du 80x86 ayant soit pour origine soit pour destination la zone mémoire contenant le code des NPcalls se traduira par une erreur sur la signature du code BP et sera donc intercepté par la vérification soit au niveau de BP, soit par un processus d'entrées/sorties intempestif.

Un masquage peut cependant se produire en cas de rupture de séquence depuis un NPcall vers un autre NPcall, soit à la suite d'une mauvaise indirection par le BP, soit à la suite d'un déséquencement dans un NPcall :

- dans le premier cas, l'erreur sera détectée si le NPcall exécuté n'a pas le même nombre de paramètres que le NPcall original car les paramètres ne sont pas signés de la même manière que les codes instructions (cf 1.6.2.1, figure 1.21),
- dans le second cas (faible probabilité), l'erreur pourra être détectée si l'état de la pile est modifié, ou si un NPcall de séquencement est touché.

Cependant, la vérification du 80x86 à travers la signature et le séquencement du processeur BP n'est pas suffisante et une vérification du flot de contrôle du 80x86 a été implantée pour compléter la vérification faite sur BP.

Pour la vérification du flot de contrôle du microprocesseur 80x86, il a été préféré de ne pas implanter un dispositif de génération de signature dérivée, d'une part pour des problèmes de compatibilité de l'analyseur de signature entre les

différents microprocesseurs utilisés dans les versions d'UCs, et d'autre part à cause du coût global d'implantation jugé a priori trop élevé compte tenu du type des microprocesseurs utilisés (pipeline en particulier), de la nature du programme système exécuté (multi-tâches, temps réel, préemptif) et du fonctionnement des NPcalls qui ne facilitent pas la gestion des références (séquencement indirect pour le 80x86).

La signature des programmes exécutés par le 80x86 est donc de type imposée ce qui permet une vérification du séquencement seul. La méthode utilisée pour l'invariance et le test de la signature sera décrite en section 4.3.3.1 . Elle reprend le principe de la vérification au niveau graphe d'état d'une machine câblée (cf 1.5) à la différence que le test est fait ici au niveau système et non plus au niveau circuit.

Les erreurs d'opération du 80x86 ne sont donc pas vérifiées directement à travers ce dispositif d'analyse de signature à l'exception des erreurs qui peuvent être détectées indirectement à travers l'analyse de signature sur le programme d'application exécuté par le BP (exécution incorrecte des NPcalls de séquencement).

La vérification du séquencement du 80x86 a été implantée pour le contrôle du graphe de l'enveloppe des tâches uniquement, ceci dans le but de compléter l'analyse de signature du code BP. Il est bien sûr possible d'étendre ce type de test à d'autres parties du système. Le principe du test est décrit dans les sections suivantes.

4.3.3.1. Application du S-codage à la vérification d'un système

Le programme à vérifier (ici le système de l'UC) est décomposé en états, une valeur arbitraire étant affectée à chacun d'eux. Lors de l'exécution des instructions correspondant à un état donné, la valeur affectée à cet état est introduite dans une signature par des instructions rajoutées à cet effet. Un dispositif matériel de génération de signature identique à celui nécessaire pour BP peut être utilisé pour la signature de NP et comme les deux processeurs ont un fonctionnement exclusif l'un de l'autre, le même matériel sert pour les deux processeurs. Par contre, l'origine des données à compacter et l'instant de compaction sont différents pour NP et BP.

L'invariance de la signature de NP peut être assurée de deux façons. La première solution consiste à choisir les valeurs (arbitraires) introduites dans la signature de telle manière que les signatures de tous les chemins arrivant au même point de jonction soient identiques. Ceci peut conduire à une situation où une même valeur correspond à plusieurs états : pour éviter cela il faut soit modifier la décomposition en états du logiciel soit utiliser des ajustements, ce qui constitue la

deuxième solution pour l'invariance de la signature de NP. Ces ajustements sont faits par le même mécanisme que pour la signature de BP.

4.3.3.2. Stockage et accès aux références

Les valeurs à compacter, les valeurs d'ajustement et les signatures de référence sont des données pour les instructions NP qui les manipulent. Elles peuvent être stockées soit comme constantes soit dans des variables.

Dans le cas des enveloppes de tâches, ces valeurs sont stockées sous forme de tableaux de constantes ayant une valeur par tâche. Il existe donc un tableau de constantes pour chaque état de l'enveloppe modifiant ou testant la signature. Dans l'état d'exécution du code application ("U_CODE" sur la figure 4.4), la valeur introduite dans la signature est en fait le résultat de la compaction de l'ensemble du code BP exécuté. Cette valeur dépend de l'application : il faut donc ajuster la signature (avec une valeur pré calculée en même temps que les références à partir du code) à chaque cycle d'exécution de l'enveloppe (Figure 4.4).

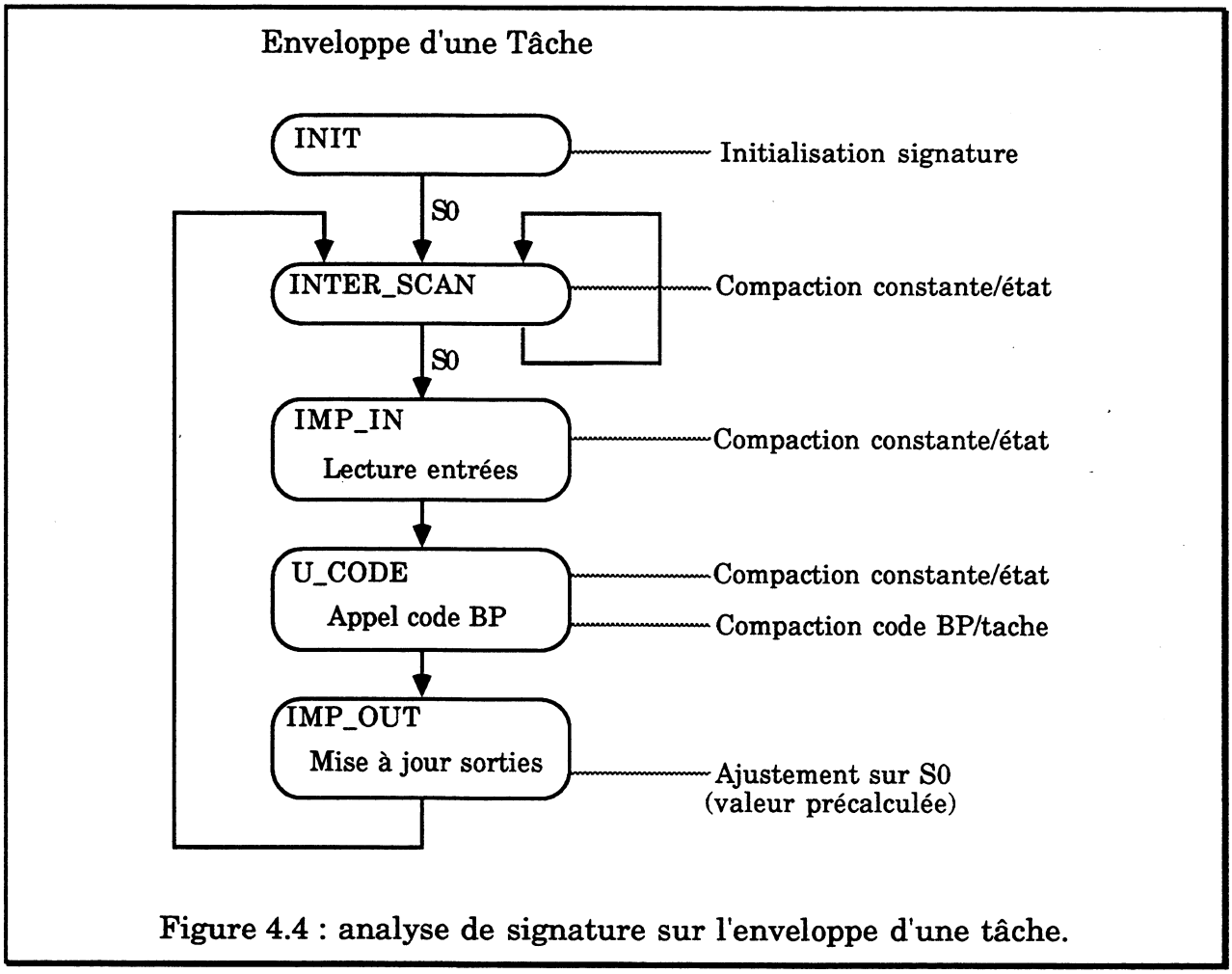


Figure 4.4 : analyse de signature sur l'enveloppe d'une tâche.

Pour une tâche donnée, le registre de signature n'est donc initialisé qu'une seule fois, lors de l'exécution de la phase "INIT" de cette tâche. Après quoi, la signature est modifiée à travers la compaction des constantes/états, l'exécution du code BP et l'ajustement sur début de cycle qui permet une invariance de la signature sur le cycle normal de fonctionnement de l'automate. En particulier, la signature n'est pas initialisée avant un appel au code BP, car sa valeur théorique est toujours la même (signature intermédiaire de l'état "U_CODE").

4.3.3.3. Génération des références

Les valeurs des constantes placées dans l'enveloppe d'une tâche pour être introduites dans la signature ou pour tester la signature en certains endroits ont été calculées et placées à la main, étant donné le faible nombre de ces valeurs.

Il est évident que dans le cas d'une extension systématique de ce type de vérification à d'autres parties du système, il serait préférable d'utiliser un programme spécifique pour extraire la structure du graphe vérifié et pour y affecter des constantes respectant l'invariance de la signature sur tous les chemins.

Dans les enveloppes des tâches, les valeurs des ajustements sur la signature de départ ne sont pas des constantes, car elles dépendent de la signature de sortie du code BP. Ces valeurs sont calculées en fin de génération des références du code BP de chaque tâche et placées dans un tableau comportant une valeur d'ajustement par tâche.

4.3.4. Sécurité sur détection d'erreur

La latence de détection des dispositifs de test de séquencement n'est pas nulle, et il faut en particulier vérifier qu'un test de signature a lieu régulièrement. Cette vérification est assurée par le fait que le rafraîchissement du chien de garde temporel est assuré dans le cycle de fonctionnement de la tâche MAST. La vérification du graphe de l'enveloppe de la tâche MAST (entre autres) permet de vérifier que le chien de garde n'est pas rafraîchi d'une manière intempestive. Cependant, dans les cas où le cycle de la tâche MAST est interrompu, il peut exister, avant le déclenchement du chien de garde temporel, des cas où une erreur est propagée sur les sorties de l'automate par un processus d'entrées/sorties intempestif. Pour éviter ce cas de figure dangereux, une sécurité a été implantée dans le PIU au niveau du dispositif d'entrées/sorties : aucun échange d'entrées/sorties n'est possible tant qu'un test positif de la signature n'a pas été effectué. Pour cela, un drapeau est positionné à chaque fois qu'un test positif de signature a lieu. Ce drapeau retombe dès qu'une autre action sur la signature est faite (instruction BP exécutée, ajustement, valeur compactée par NP, écriture du registre de signature). Lors d'un échange d'entrée/sorties, ce drapeau est testé et

s'il est faux, une erreur est détectée et le processus d'entrées/sorties stoppé. Ce drapeau est accessible dans un registre particulier du PIU et il est donc possible de tester régulièrement qu'il n'est pas "collé" à la valeur qui autoriserait en permanence les échanges d'entrées/sorties.

Avec ce mécanisme de blocage, un test de signature doit avoir lieu avant chaque processus d'entrées/sorties (cf 4.3.2.3). Pour cette raison, les NPcalls d'E/S explicites testent la signature, avec une signature de référence placée dans la table des références de l'analyse de signature du code BP. Sur l'enveloppe d'une tâche, la signature devra donc être testée une fois avant la phase de lecture des entrées et une autre fois avant la phase de mise à jour des sorties.

Le test avant lecture des entrées est effectué par NP avec une référence placée dans un tableau de constantes comprenant une référence par tâche.

Le test avant mise à jour des sorties est assuré par l'analyse du code BP lors de l'exécution du NPcall qui marque la fin de la phase d'exécution du code BP.

Entre ces deux tests (exception faite des NPcall E/S explicites) toute opération d'E/S est interdite par un blocage du picoséquenceur du PIU (Figure 4.5).

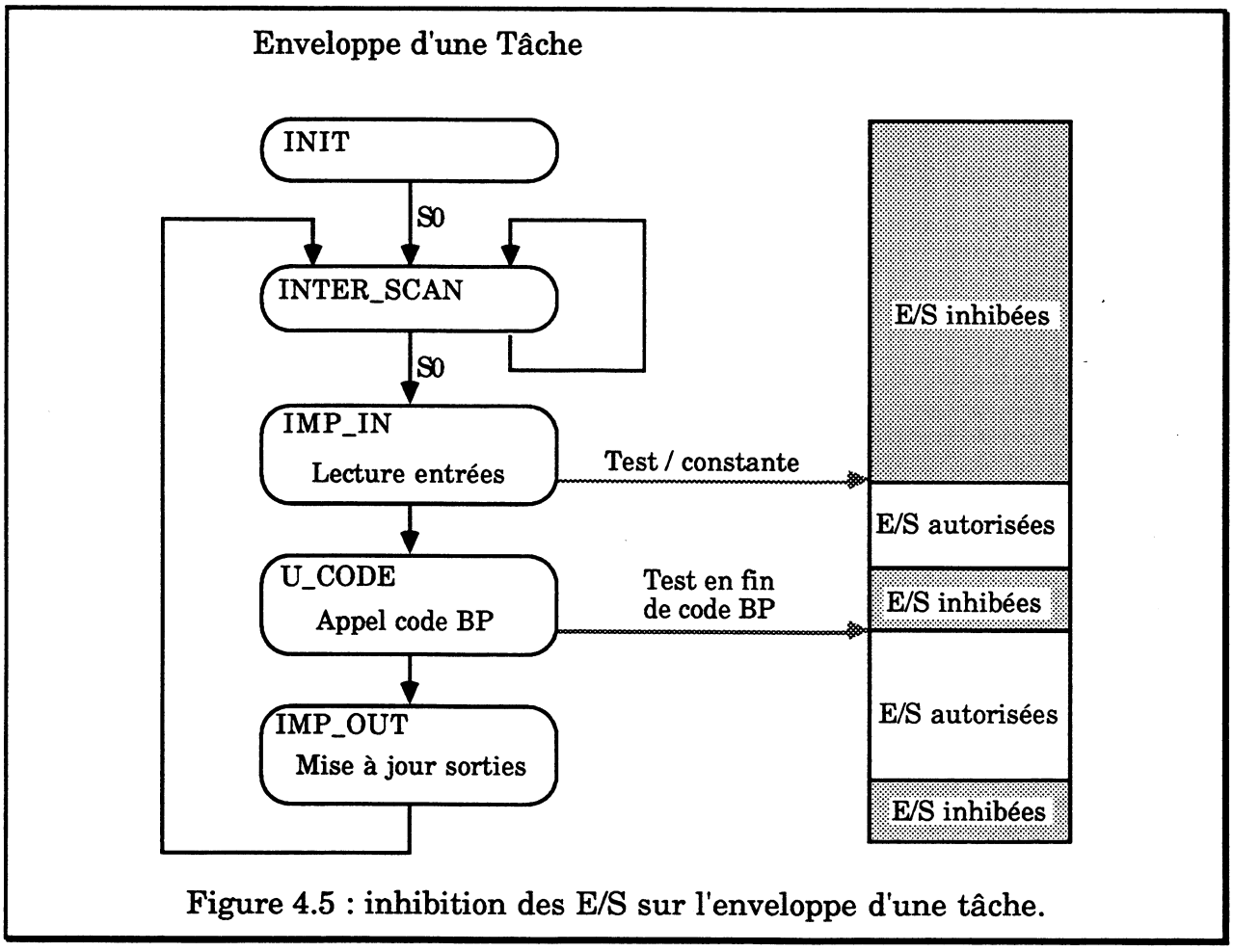


Figure 4.5 : inhibition des E/S sur l'enveloppe d'une tâche.

4.3.5. Fonctionnement multi-tâches

4.3.5.1. Changement de contexte

Le fonctionnement multi-tâches d'une UC implique qu'une tâche d'un niveau de priorité donné puisse interrompre le cycle d'une tâche moins prioritaire, y compris pendant la phase d'exécution du code BP. Lors d'un changement de tâche, différentes informations relatives à l'analyse de signature doivent être préservées :

- le contenu du registre de signature,
- la valeur du pointeur de référence,
- le drapeau d'autorisation des échanges d'entrées/sorties (valeur "vraie" si un test positif de signature vient d'avoir lieu).

Ces informations sont conservées avec les autres informations relatives aux contextes des tâches.

Pour la restauration de la signature, le registre du dispositif de compaction implanté dans le PIU peut être chargé directement.

Pour la restauration du drapeau d'autorisation des échanges d'entrées/sorties, une écriture directe n'est pas nécessaire. En effet, toute modification du registre de signature place ce drapeau à la valeur "faux". Un test de signature correct au contraire le place à la valeur "vraie". En conséquence :

- si le drapeau à restaurer a la valeur "vraie", il suffit de tester la signature avec la valeur que l'on vient d'écrire dans le registre de signature, ce qui met le drapeau dans la bonne position,
- Si ce drapeau à restaurer a la valeur "faux", alors le simple fait d'avoir rechargé le registre de signature lui a déjà donné la valeur "faux" et il n'y a donc rien de plus à faire.

La restauration du pointeur de référence n'appelle pas de commentaires particuliers, il s'agit d'une variable du système.

4.3.5.2. Gestion des références

La structure des données pour la gestion des références est indiquée sur la figure 4.6 .

Pour chaque tâche, il est nécessaire de connaître le pointeur sur la première référence du code BP de la tâche ainsi qu'une valeur permettant d'ajuster la signature après l'exécution du code BP sur la valeur de la signature intermédiaire de l'état de départ d'un cycle de tâche. Ces deux informations sont calculées par le programme de génération des références du code BP et sont stockées en tout début de la table des références. Deux tableaux de sept valeurs chacun (7 tâches) sont placés avant les références.

Le pointeur de début des références d'une tâche est placé dans le pointeur de références dans l'état d'exécution du code BP de la tâche, juste avant le lancement du BP.

L'ajustement de la signature sur la valeur de début de cycle est réalisé juste après la phase de mise à jour des sorties d'une tâche, c'est à dire en fin de cycle, avant le rebouclage dans l'état d'attente du signal de reprise du cycle.

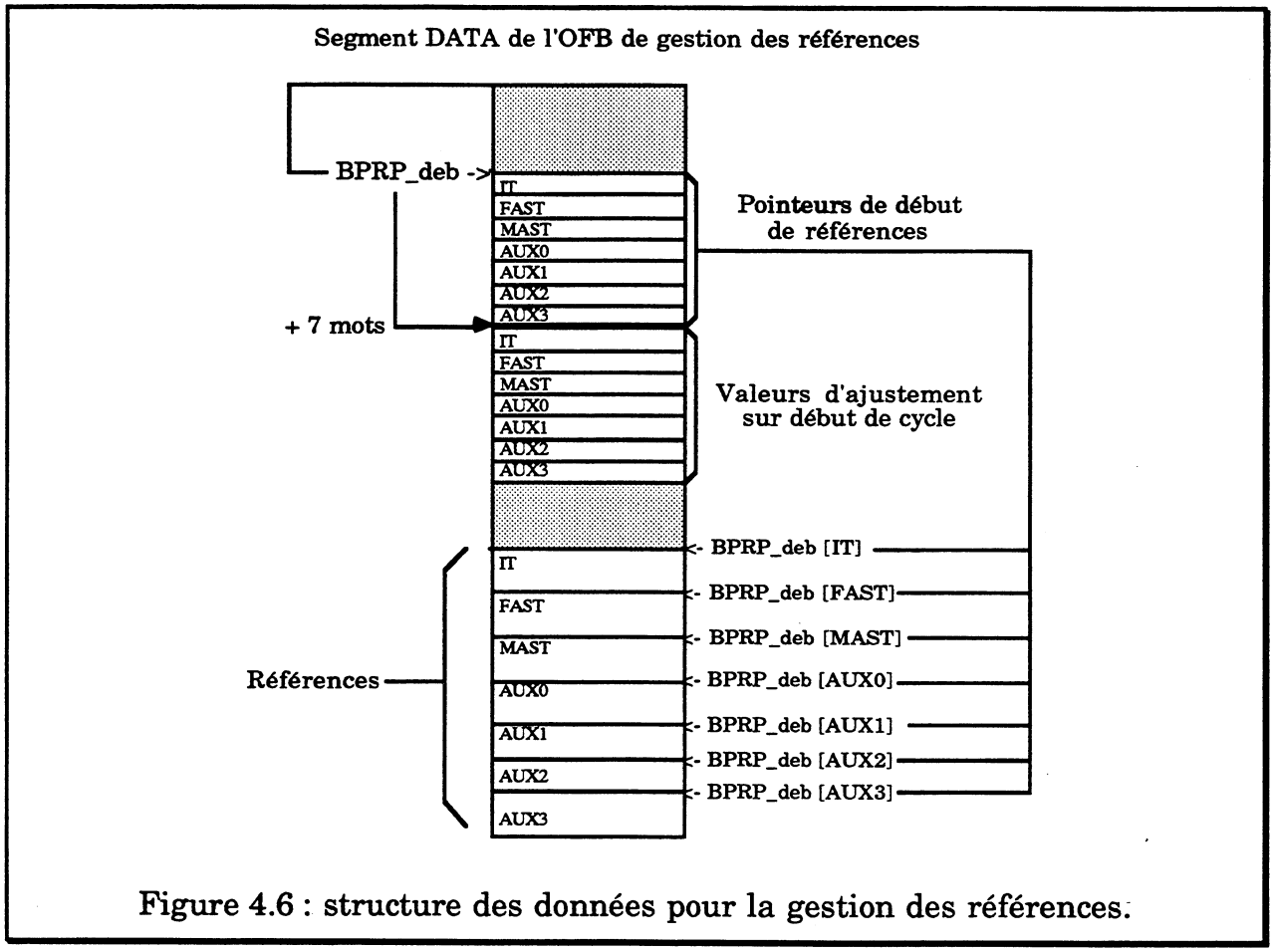


Figure 4.6 : structure des données pour la gestion des références.

4.4. Implantation des dispositifs de test

4.4.1. Organisation générale des dispositifs implantés

Les modifications apportées à l'architecture d'UC étudiée pour qu'elle supporte le schéma de test en ligne décrit sont de deux types. Il s'agit en premier lieu d'une évolution matérielle au niveau du PIU et en second lieu d'une évolution logicielle du système.

Les modifications matérielles concernent l'ajout dans le PIU d'un dispositif de génération et de test de signature directement relié au processeur booléen et d'un mécanisme de blocage des E/S greffé au dispositif de gestion du bus d'entrées/sorties.

Les modifications logicielles portent essentiellement sur certains NPcalls, sur l'enveloppe des tâches et sur le noyau temps réel du système (changement de tâches).

La figure 4.7 résume l'implantation des différents éléments du dispositif de test de séquencement au niveau des constituants de l'UC :

- PIU : dispositif matériel décrit en section 4.4.2 ,
- Code système : quelques instructions pour gérer le matériel rajouté,
- Données système : le pointeur (BPRP) de la référence courante,
- Données OFB : la table des références.

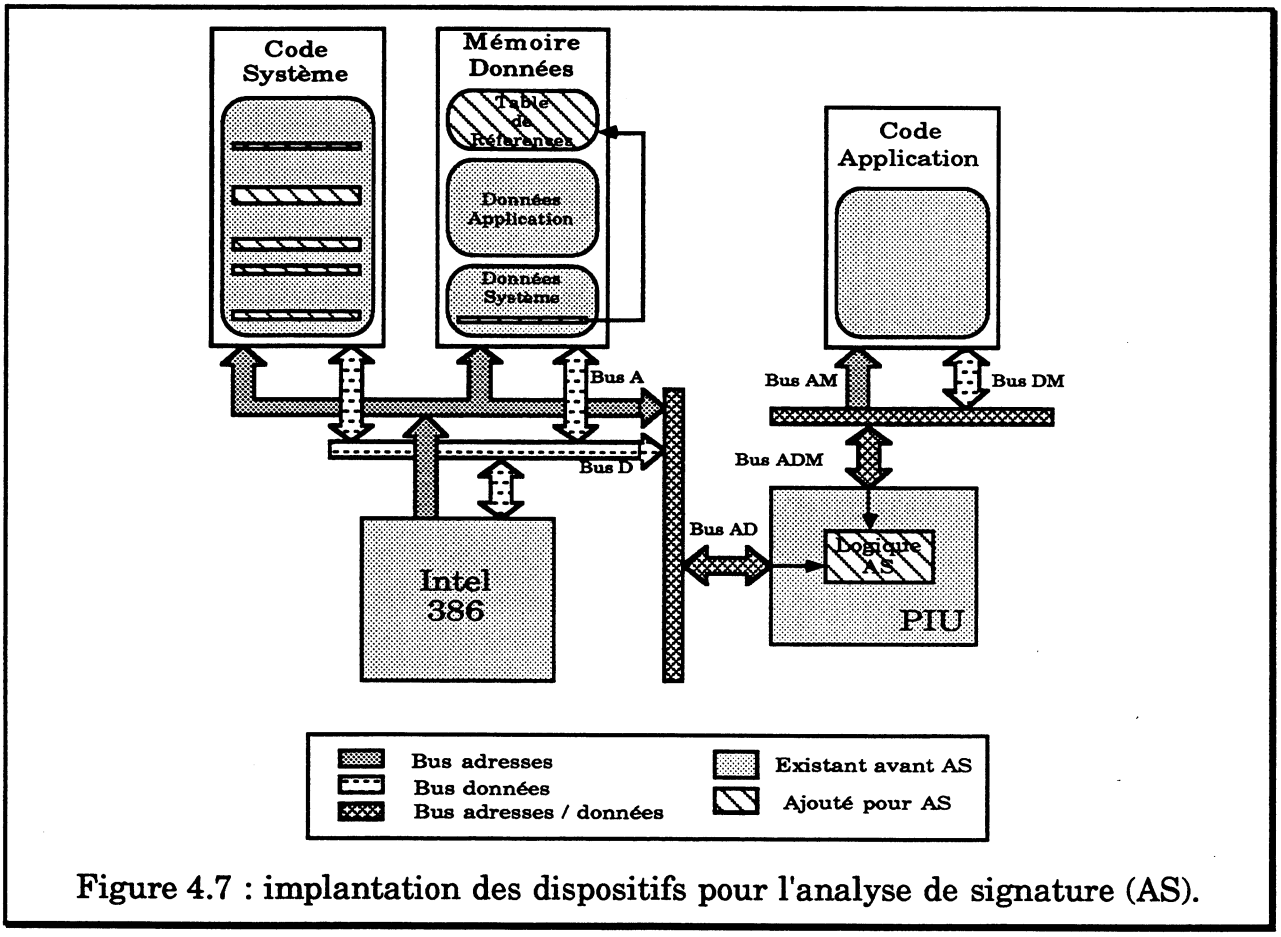


Figure 4.7 : implantation des dispositifs pour l'analyse de signature (AS).

4.4.2. Implantation matérielle

La figure 4.8 décrit l'emplacement et les interconnexions des dispositifs matériels de l'analyseur de signature,par rapport à l'organisation du PIU.

Les dispositifs matériels nécessaires ont tous été intégrés dans le composant ASIC PIU. Ce matériel se décompose en trois parties essentielles :

- un dispositif de compaction pour la génération de la signature commune aux processeurs BP et 80x86,
- un dispositif de contrôle de l'analyse de signature,
- un dispositif de blocage des échanges d'entrées/sorties.

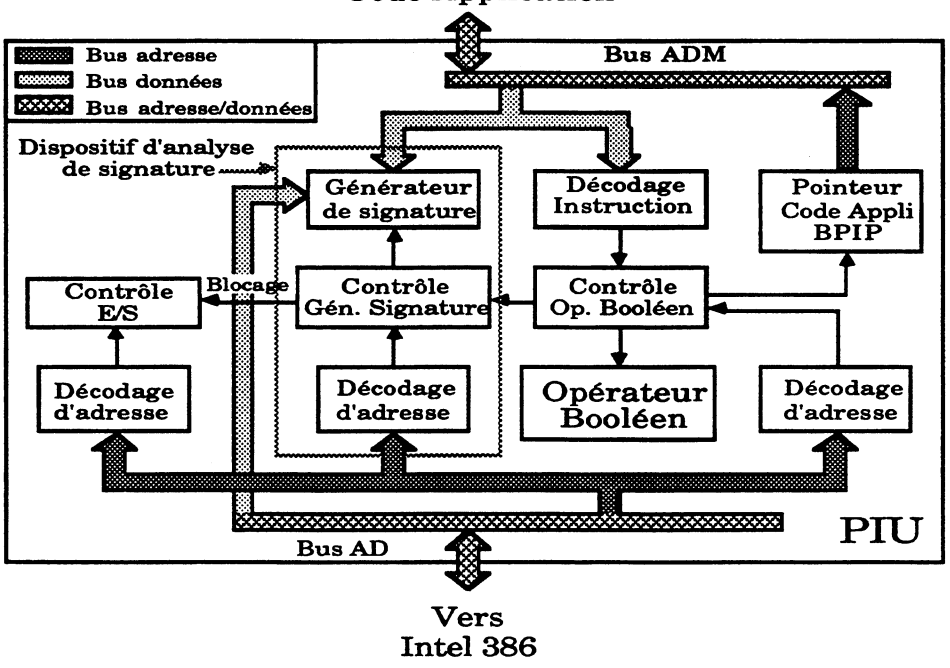


Figure 4.8 : intégration des dispositifs matériels au PIU.

4.4.2.1. Dispositif de génération de signature

Le dispositif de génération de la signature est composé classiquement d'un MISR à OU Exclusifs internes (cf 1.2.2.1) comportant 16 entrées parallèles. Le polynôme diviseur est câblé. Il est primitif, de degré 16 et possède seulement 4 coefficients à 1 (bits 15,11,4,0). Ceci limite le nombre de portes OU Exclusif à implanter pour ce polynôme.

Le schéma générique pour un bit de ce dispositif de compaction est présenté sur la figure 4.9 .

Les OU Exclusif 1, 2 et 3 ont le rôle suivant :

- OU Exclusif 1 : entrée d'une donnée à compacter,
- OU Exclusif 2 : masque pour la compaction des paramètres,
- OU Exclusif 3 : rebouclage du MISR.

Le tableau 4.1 indique, pour chaque bit du dispositif de compaction, la présence ou non de ces OU Exclusifs.

L'entrée du MISR est reliée à un multiplexeur (commande CRDFIP) qui permet de sélectionner :

- CRDFIP=1 : le bus donnée mémoire "DM" pour la compaction des paramètres de NPcall lus par le 80x86 à travers le pointeur d'instruction BP,

- CRDFIP=0 : la sortie du registre appelé "PIPE" qui sert pour la lecture anticipée des instructions BP et pour l'écriture par le 80x86 de données dans le PIU.

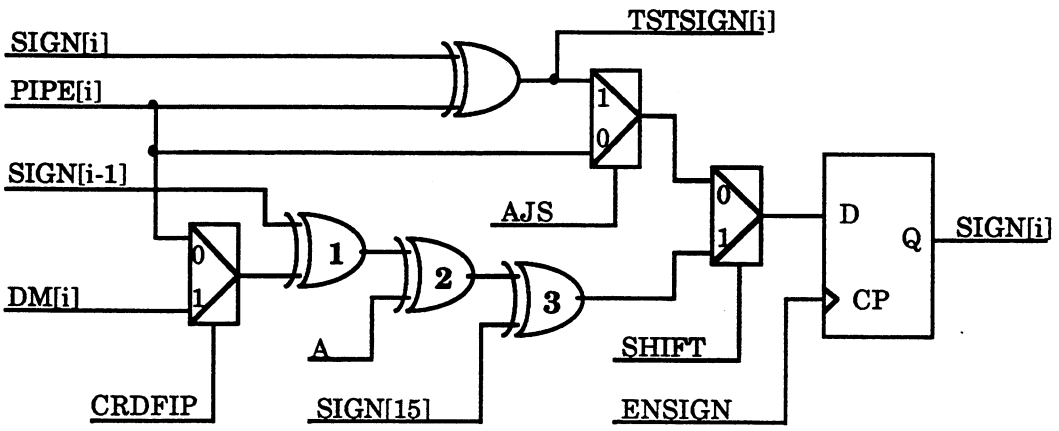


Figure 4.9 : schéma générique pour un bit du dispositif de génération de signature.

Un second multiplexeur (commande SHIFT) permet de charger directement le registre de signature (SHIFT=0). La valeur chargée est sélectionnée par un troisième multiplexeur (commande AJS) qui permet soit de faire un ajustement de signature (AJS=1) en réalisant l'opération SIGN <= SIGN ⊕ PIPE, soit de charger le registre de signature : SIGN <= PIPE.

Tableau 4.1 : présence des portes OU Exclusif 1, 2 et 3 pour chaque bit du dispositif de compaction.

Bit	XOR 1	XOR 2 (entrée A)	XOR 3
0			X
1,2,3	X		
4	X		X
5,6	X		
7	X	CRDFIP	
8,9	X		
10	X	PARAM	
11	X		X
12,13,14	X		
15	X		X

Deux masques sont placés en entrée du MISR. Un masque (0080) est utilisé lorsqu'un paramètre de NPcall est lu (CRDFIP=1) et le second masque (0400) est utilisé pour la compaction de paramètres de certaines instructions booléennes (PARAM=1). Aucun masque n'est actif lorsque la donnée compactée est une instruction booléenne normale ou une valeur émise par le processeur numérique.

Les différentes fonctionnalités du dispositif de compaction sont résumées dans le tableau 4.2 .

Tableau 4.2 : fonctionnalités du dispositif de compaction.

SHIFT	AJS	CRDFIP	SIGN
0	0	Ø	PIPE
0	1	Ø	PIPE ⊕ SIGN
1	Ø	0	Comp(PIPE)
1	Ø	1	Comp(DM)

(la fonction Comp est la compaction d'une valeur par le MISR)

4.4.2.2. Dispositif de contrôle de l'analyseur de signature

Le dispositif de contrôle de l'analyseur est composé de trois éléments. Une partie de logique combinatoire sert à élaborer les signaux de commande du dispositif de génération de signature, une seconde partie sert à tester la signature, et une troisième partie, composée de deux registres, sert à valider et mémoriser les sources d'erreurs.

Contrôle du dispositif de compaction

Ce dispositif contrôle les commandes des multiplexeurs et la validation du registre de signature. Trois actions au niveau du PIU provoquent une modification du registre de signature :

- écriture d'une valeur par NP à une des adresses du PIU suivantes :
  - AJS : ajustement avec une valeur émise par le 80x86,
  - CPCT : compaction d'une valeur émise par le 80x86,
  - SIGN : écriture directe par le 80x86 du registre de signature,
- lecture par NP d'un paramètre de NPcall,
- lecture par BP d'un mot du code BP.

Chacune de ces trois actions provoque une modification du registre de signature et remet à zéro le drapeau TESTOK utilisé pour le blocage des E/S.

Mécanisme de test de la signature

Un test de signature est déclenché par l'écriture d'une valeur par le 80x86 à l'adresse TESTS du PIU. La comparaison est effectuée par un NOR à 16 entrées reliées à la sortie du OU Exclusif dont une entrée vient du registre de signature et l'autre du registre PIPE. Le résultat de cette comparaison est placé dans le drapeau TESTOK.

Registre de validation des sources d'erreurs : CONFS

Le registre CONFS comporte 3 bits dont la signification est la suivante :

bit 0 = 1 : validation du blocage des E/S et interruption si blocage,

bit 1 = 1 : déclenchement d'une interruption sur test de signature négatif,

bit 2 = 1 : activation du chien de garde général de l'UC si erreur de signature ou blocage des E/S.

Les trois bits du registre CONFS sont toujours remis à zéro par un reset du PIU. Lorsque ces trois bits sont à zéro, alors aucune erreur relative à l'analyse de signature n'est signalée, et le composant se comporte comme un PIU standard.

Registre de mémorisation des sources d'erreur : STATS

Le registre STATS comporte 3 bits dont la signification est la suivante :

bit 0 = 1 : une interruption a été déclenchée suite à un blocage d'E/S. Ce bit est remis à zéro par l'écriture d'un 1 dans le bit 0 du registre CONFS.

bit 1 = 1 : un test de signature faux a eu lieu. Cette erreur est signalée conformément à la programmation du registre CONFS. Ce bit est remis à zéro par l'écriture d'un 1 dans le bit 1 du registre CONFS.

bit 15 =1 : drapeau TESTOK : indiquant qu'un test de signature correct vient d'avoir lieu et qu'aucune action sur la signature n'a eu lieu depuis.

4.4.2.3. Dispositif de blocage des E/S

Le dispositif de blocage des E/S est relié directement au dispositif de contrôle du bus d'entrées/sorties (Figure 4.10).

Pour détecter l'occurrence d'un échange d'entrées/sorties, un bit particulier du bus de contrôle des entrées/sorties ("EXCHIO") est testé. Ce bit est positionné à 1 lors d'un échange d'E/S. Pour déterminer si les échanges d'E/S sont autorisés, le

drapeau "TESTOK" est testé. Ce drapeau est à 1 lorsqu'un test positif de signature vient d'avoir lieu. Il est remis à 0 dès qu'une action modifiant le registre de signature a lieu. Les E/S ne sont autorisées que si le drapeau TESTOK est à 1 ou si le dispositif de blocage des E/S n'est pas validé.

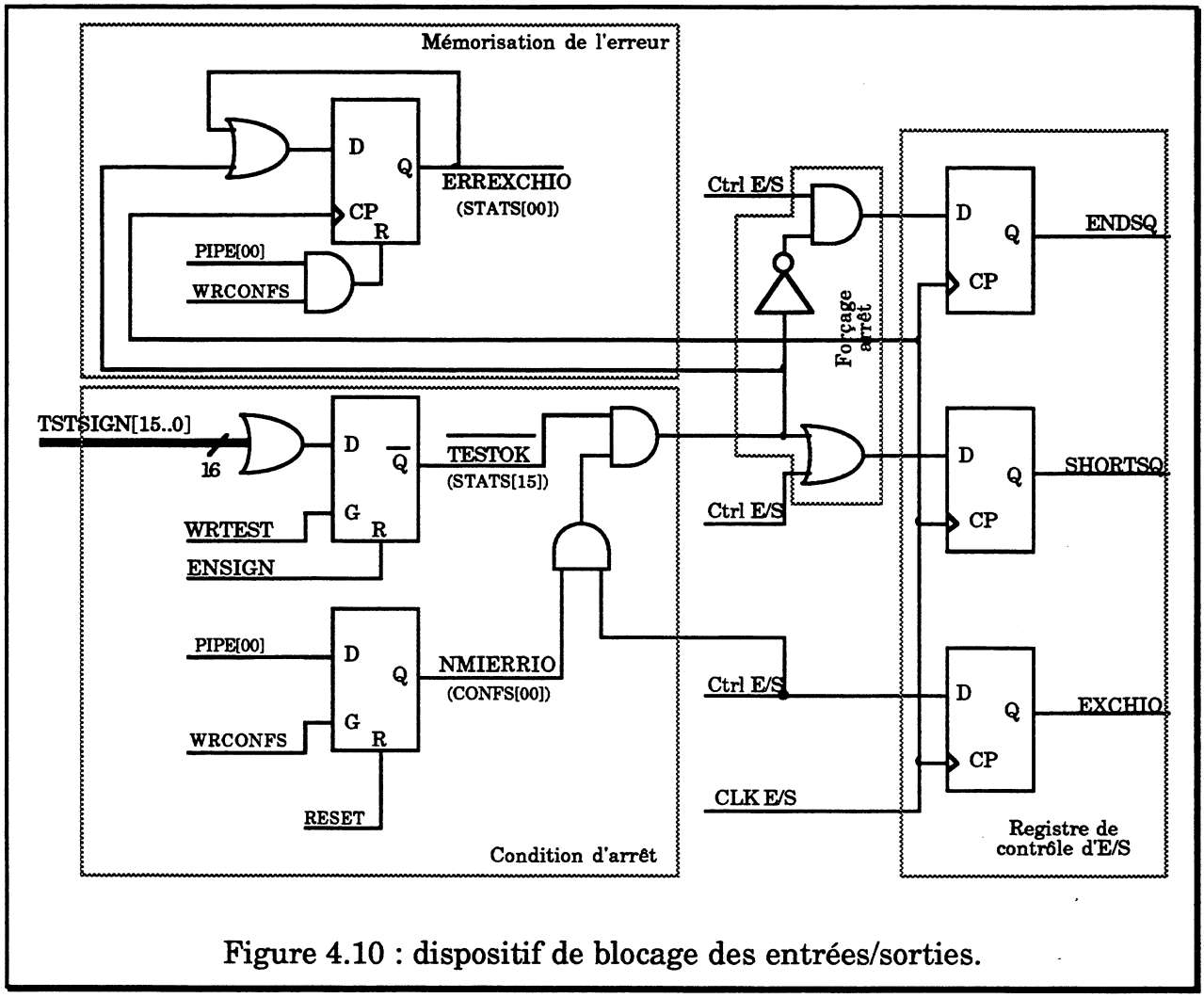


Figure 4.10 : dispositif de blocage des entrées/sorties.

Le blocage des E/S est validé par la présence d'un 1 dans le bit zéro du registre CONFS. Dans ce cas, si le signal EXCHIO doit passer à 1 sur le bus de contrôle des entrées/sorties alors que le drapeau TESTOK est nul, un arrêt de l'échange est imposé par le forçage des signaux adéquats ("SHORTSQ" et "ENDSQ"). Dans ce cas, l'erreur est mémorisée dans le bit zéro du registre STATS. Ce drapeau d'erreur est automatiquement remis à 0 dès que le blocage des E/S est revalidé.

4.4.3. Implantation logicielle

Les NPcall ayant été modifiés pour le fonctionnement de l'analyse de signature sur le code BP sont les NPcalls de séquencement, c'est à dire ceux qui modifient le pointeur du code application (BPIP). Les NPcall d'échanges d'E/S explicites ont

également été modifiés afin que la signature soit testée avant l'échange d'E/S, ceci afin que le mécanisme de blocage ne se déclenche pas. Les NPcall "Début de code" et "Fin de code" réalisent eux aussi un test de la signature. Ce sont les seuls points de test obligatoires de la signature au niveau du code BP.

Le fonctionnement de l'analyseur pour chaque type de NPcall modifié est très simple. Il correspond directement à l'implantation logicielle des traitements du moniteur DJAM tels qu'ils sont décrits en section 3.2.6 . Pendant l'exécution d'un NPcall ayant une action sur la signature, l'accès aux références et la gestion du pointeur de références sont pris en charge par le 80x86. Les valeurs ainsi obtenues sont ensuite écrites aux adresses AJS et TESTS du PIU.

Pour l'enveloppe des tâches, la signature est générée par une écriture, à l'adresse CPCT du PIU, de valeurs constantes attribuées à chaque état d'un cycle de tâche. L'ajustement en fin d'exécution du code BP est fait par une écriture à l'adresse AJS du PIU avec une valeur trouvée dans la table des références.

4.5. Efficacité des mécanismes implantés

L'efficacité d'une vérification de flot de contrôle par analyse de signature dérivée a déjà été présentée à plusieurs reprises dans la littérature, de même que l'efficacité de différents mécanismes classiques de test en ligne. Les approches utilisées peuvent être réparties en plusieurs catégories :

- injections de fautes sur un système à base de microprocesseur et évaluations de plusieurs moyens de détection au niveau système : [Schm 82], [Damm 86], [Gunn 89], [Chil 89], [Karl 91], [Mire 92], [Kana 92],
- injections de fautes sur un système disposant d'un mécanisme d'analyse de signature dérivée pour la vérification du flot de contrôle d'un microprocesseur : [Schu 86], [Schw 87], [Made 91b],
- simulations de défauts internes dans un microprocesseur et évaluations de l'impact au niveau système : [Duba 88], [Czec 90], [Ohls 92].

Cependant, l'architecture spécifique du prototype d'UC considérée et la manière dont les dispositifs de test par analyse de signature y ont été implantés ne permettent pas de faire une extrapolation directe de la plupart de ces chiffres, et en particulier de ceux relatifs à l'efficacité d'une analyse de signature dérivée sur un microprocesseur standard. C'est pourquoi une évaluation approfondie a été menée sur le prototype d'UC étudié, d'une part, pour valider l'implantation des dispositifs rajoutés et d'autre part, pour pouvoir chiffrer l'efficacité relative des mécanismes de test en ligne présents sur le prototype d'UC étudié.

La première évaluation a été faite par modélisation du comportement de l'UC en cas de défaut interne à cette UC et sera présentée en section 4.5.1 .

Un deuxième type d'évaluation a été faite par des expériences d'injection de fautes sur le prototype d'UC réalisé et sera décrit dans les sections 4.5.2 à 4.5.4 .

Une comparaison entre les deux évaluations sera faite en section 4.5.5 .

4.5.1. Modélisation d'un taux de détection global au niveau d'une UC

L'évaluation de dispositifs de détection d'erreurs au niveau système, dans une architecture utilisant des moyens de détection d'erreurs autres que la réplication de composants, n'a quasiment jamais été abordée d'une autre manière que par l'expérimentation, c'est à dire par injection de fautes, sur un système réel ou sur un modèle de système simulé [1].

La quasi totalité des études théoriques concernant l'évaluation du niveau de sûreté d'un système sont faites sur des systèmes massivement redondants. Dans ce type d'étude, la défaillance de composants du système est considérée mais rarement leurs modes de défaillance. Or un circuit VLSI, lorsqu'il est défaillant, ne se contente généralement pas de s'arrêter purement et simplement de fonctionner : son comportement est dans la plupart des cas totalement imprévisible et peut donner lieu à des phénomènes latents.

Les modèles utilisés dans le cas de systèmes massivement redondants, qui prennent essentiellement en compte les taux de défaillance des composants, sont difficilement exploitables pour une évaluation du niveau de sûreté d'un système dans lequel la détection est assurée par des moyens de test en ligne à faible coût.

Pour le cas du prototype d'UC étudié, une modélisation de la propagation des erreurs dans le système a été faite. Ce modèle n'utilise pas directement la notion de taux de défaillance. Au contraire, l'état de départ de ce modèle correspond à un système dans lequel une faute vient de se manifester sous forme d'une erreur.

4.5.1.1. Principe de la modélisation

Lorsque plusieurs mécanismes de détection d'erreurs sont présents dans un même système, l'évaluation de l'efficacité combinée de ces mécanismes (taux de détection global d'erreurs au niveau du système) pose un problème difficile à résoudre.

---

[1] Le terme "simulation de fautes" n'est volontairement pas employé car il est utilisé dans le cas de l'évaluation de la couverture d'un jeu de vecteurs de test, ce qui n'a aucun rapport avec le cas présent.

Pour chaque mécanisme il est possible de connaître ou d'estimer assez précisément un taux de couverture par rapport au modèle des erreurs pouvant être détectées par ce mécanisme. Cependant, lorsqu'il existe plusieurs mécanismes, ayant les latences de détection différentes et dont les modèles d'erreurs détectées sont également différents, il n'est pas possible d'ajouter simplement les taux de détection individuels de chaque mécanisme dans la mesure où certaines erreurs peuvent être détectées par plusieurs mécanismes à la fois. La détection interviendra, dans ce cas, par le mécanisme dont la latence est la plus faible, dégradant ainsi l'efficacité potentielle des mécanismes dont la latence est supérieure.

Pour connaître l'efficacité globale de la détection de plusieurs mécanismes combinés, une modélisation de la propagation des erreurs dans le système est nécessaire. La figure 4.11 présente un exemple de modèle pouvant être utilisé pour chiffrer un taux de détection global au niveau d'un système.

Ce modèle est constitué par un graphe orienté dans lequel chaque état représente soit :

- l'occurrence d'une faute, d'une erreur ou d'un mode de défaillance du système,
- un mode de fonctionnement du système (fonctionnement normal ou détection d'une erreur).

La propagation d'une erreur est représentée par un chemin commençant toujours par l'occurrence d'une faute (événement racine). Une faute survenant pendant un mode de fonctionnement donné du système engendre un type d'erreur particulier qui pourra être soit détecté soit propagé par d'autres modes de fonctionnement du système. Dans le modèle, la propagation d'une erreur s'arrête soit parce qu'elle est détectée, soit parce que la modélisation du comportement du système en présence de cette erreur n'est plus possible. Les deux types d'états n'ayant aucun successeur sont donc :

- les états correspondant à la détection d'une erreur,
- les états correspondant à un mode de défaillance du système.

Chaque arc du modèle est pondéré par une probabilité qui est calculée en fonction de la structure et de l'activité du système. Des mesures, effectuées sur une UC, ont ainsi permis d'obtenir les moyennes et les plages de variation de certains paramètres de son activité.

La somme des probabilités des arcs sortant d'un état est toujours égale à un, et le modèle peut donc être représenté par une matrice stochastique, comme un modèle de Markov en "temps discret".

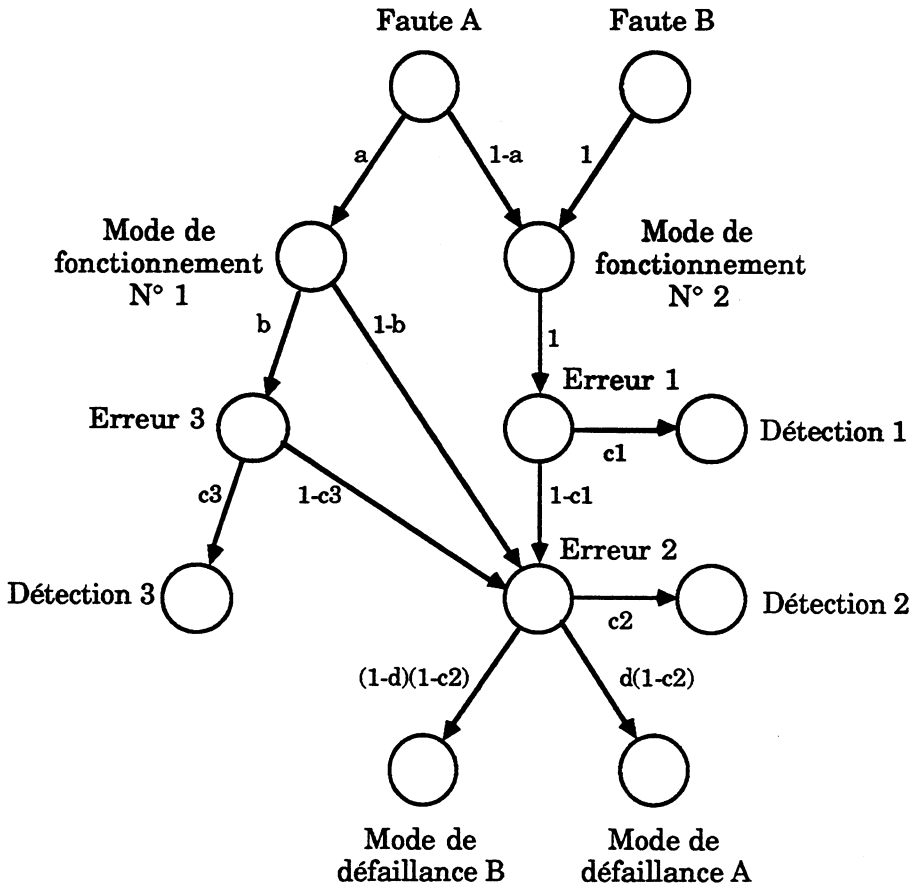


Figure 4.11 : modélisation d'un taux de détection global.

Les mécanismes de détection du système sont associés à certains arcs du modèle. Ces arcs sont issus d'états correspondant à l'occurrence du type de défaillance ou d'erreur détectable par un mécanisme donné et tous ont pour destination un état du modèle n'ayant aucun successeur. Ces arcs sont pondérés par le taux de détection local de ce mécanisme. Le calcul de ce taux de détection local est fait en fonction de l'état correspondant à l'erreur et du mécanisme considéré. Par exemple :

- un contrôle de parité détecte 50% de l'ensemble des erreurs sur un bus de données,
- une vérification des codes illégaux détecte une erreur de lecture du code d'un programme dans la proportion des codes illégaux du jeu d'instruction d'un processeur.

Pour certains mécanismes, l'état correspondant à l'erreur représente directement le modèle d'erreur détecté par un mécanisme. Dans ce cas, le taux de détection local de ce mécanisme est égal à 1 .

La latence de détection d'un mécanisme est prise en compte par le nombre d'états entre l'événement racine et l'état de détection associé à ce mécanisme. Par exemple, dans le modèle de la figure 4.11 , une erreur provoquée par le type de faute A peut être détectée par l'un des trois mécanismes mais les mécanismes 1 et 3 ont une latence de détection inférieure au mécanisme 2.

Chaque chemin entre un événement racine et un état de détection ou de défaillance est pondéré par le produit des probabilités des arcs de ce chemin. La probabilité d'arriver dans un état sans successeur est égale à la somme des probabilités de tous les chemins conduisant dans cet état.

La probabilité d'occurrence d'un état de détection donné est ainsi le taux de détection au niveau système du mécanisme correspondant à cet état.

Le taux de détection global du système est donc la somme des probabilités de tous les états de détection.

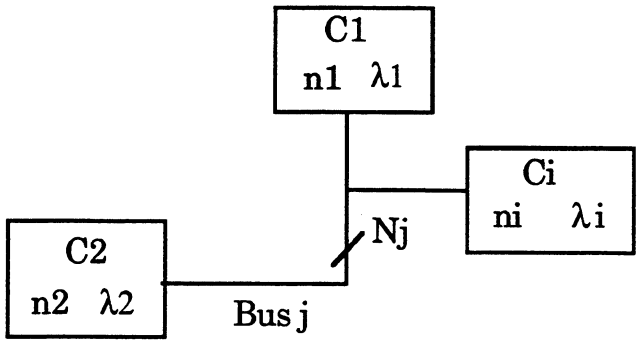
On pourra remarquer que dans ce type de modélisation, une erreur n'est jamais masquée. En effet, la propagation d'une erreur se traduit toujours soit par une détéction soit par l'occurence d'un mode de défaillance de l'UC. Pourtant, parmi les erreurs se produisant dans un système, certaines disparaissent sans être dectéctées et sans donner lieu à une défaillance du système [Chil 89]. Vis à vis de ces erreurs, la modélisation est donc un pire cas. Par contre, il existe également des erreurs qui ne sont pas détectées mais qui, en présence d'autres erreurs peuvent se combiner pour aboutir à une inhibition de certains mécanismes de test ou (et) à une défaillance du système. Ce problème d'accumulation des fautes n'est pas du tout pris en compte dans la modélisation présentée ici. Les erreurs sont donc supposées indépendantes.

Les résultats obtenus avec ce type de modèle dépendent en grande partie des fautes considérées dans les événements racines. Dans le cas de la modélisation d'une UC, les fautes qui ont été considérées sont des défauts transitoires (un cycle mémoire) sur les bus d'adresses et de données de l'UC. Cette hypothèse de fautes correspond par exemple à un parasitage du système qui affecte un échange entre deux composants. D'une manière plus générale, on considère que, pour être propagée dans le système, une erreur due à une faute dans un composant doit "sortir" de ce composant, donc se manifester sur un des bus de celui-ci.

Le calcul de la proportion de fautes apparaissant sur chaque bus (probabilité des événements racines) a été fait de la manière suivante :

A chaque bus, on affecte un taux de défaillance (λ_{busj} sur la figure 4.12) calculé à partir du taux de défaillance par signal ($\lambda i/ni$) des composants (Ci)

connectés sur ce bus et du nombre de signaux de ce bus (Nj). Comme dans [Edmo 90], les bus connectés à de nombreux composants ou à des composants peu fiables sont supposés être le plus souvent en défaut. A partir des chiffres ainsi obtenus, une répartition des erreurs sur les différents bus de l'UC est calculée (Taux d'erreur sur bus j), puis affectée aux événements racines. Il s'agit donc de valeurs permettant de caractériser l'occurrence de défauts, sur les différents bus, d'une manière relative.



$$\lambda_{bus\,j} = Nj \sum_{i} \lambda_i / ni$$

$$\text{Taux d'erreur sur bus j} = \frac{\lambda_{bus\,j}}{\sum_{i} \lambda_{bus\,i}}$$

Figure 4.12 : principe du calcul de la probabilité des événements racines.

Le modèle utilisé pour calculer le taux de détection d'une UC ne sera pas détaillé ici car il demande une description beaucoup trop approfondie de la structure et du fonctionnement d'une UC. Seuls les résultats obtenus seront présentés dans la section suivante.

4.5.1.2. Résultats de la modélisation

Le modèle réalisé pour une UC a été exploité avec trois différentes configurations de mécanismes de test en ligne :

- version d'UC avec les mécanismes de base décrits en section 4.2.3 . Cette version est utilisée comme référence,
- version d'UC comprenant, en plus des mécanismes de base, le dispositif d'analyse de signature décrit en section 4.3 ,
- version d'UC comprenant, en plus des mécanismes de base et du dispositif d'analyse de signature, une vérification de parité sur l'ensemble des zones mémoire, y compris le code système.

Les résultats sont présentés sur la figure 4.13 .

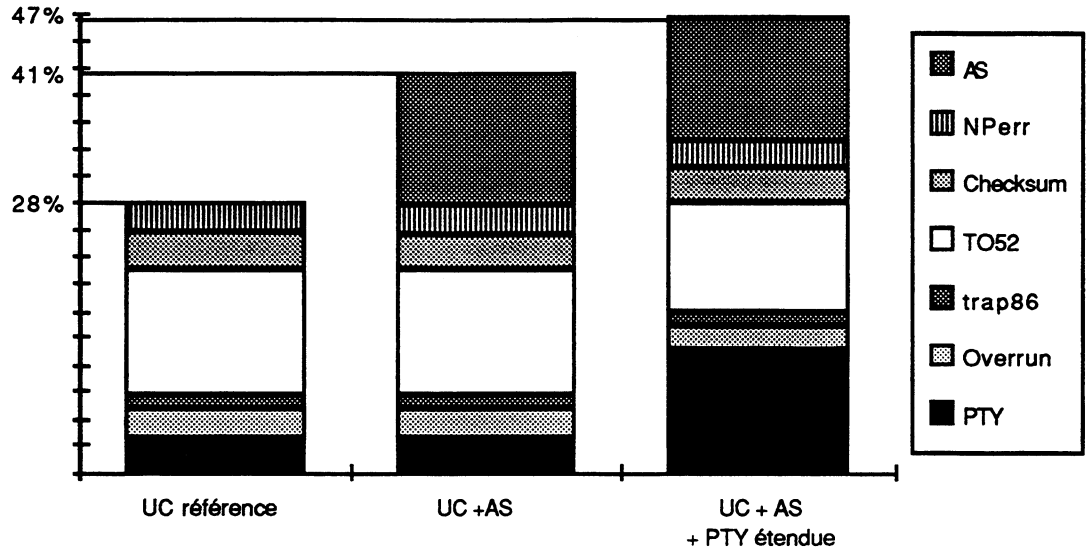


Figure 4.13 : comparaison du taux de détection modélisé.

Sur la figure 4.13, on peut constater une augmentation assez nette du taux de couverture global grâce à la présence de l'analyse de signature pour la vérification du flot de contrôle de l'UC. En comparaison, l'amélioration apportée par une extension de la parité à l'ensemble des zones mémoire est bien moins interresante si l'on considère le rapport entre le coût d'implantation et l'efficacité.

Les résultats obtenus par cette modélisation ne peuvent évidemment pas être considérés en valeur absolue mais servent uniquement de comparaison entre différentes combinaisons de mécanismes de test en ligne. L'amélioration apportée par l'analyse de signature est, en ce sens, assez nette.

Le même modèle a été utilisé pour faire une étude de la sensibilité du taux de détection par rapport à certains paramètres. Entre autres, la sensibilité au taux d'occupation mémoire dans l'espace cartouche a été évaluée en présence d'analyse de signature sur le code application (zone cartouche). Sur la figure 4.14 , seuls les mécanismes dont le taux de détection varie en fonction du taux d'occupation mémoire ont été représentés, c'est à dire la parité (PTY), les codes de NPcalls inexistants et l'analyse de signature.

On note, sur ma figure 4.14, que la détection par parité diminue sensiblement quand le taux d'occupation mémoire augmente, ce qui est normal (cf 4.2.3.1). On remarque également que le taux de détection global ne varie par contre absolument pas quand l'analyse de signature est présente, le taux de détection de ce mécanisme augmentant dans la même proportion que le taux de détection par parité diminue.

Ceci vient du fait que toutes les erreurs détectables par la parité le sont aussi par l'analyse de signature (l'inverse n'est pas vrai).

Pour une erreur détectable par la parité et quand l'analyse de signature est présente, la parité détecte l'erreur avant l'analyse de signature car elle présente une latence de détection inférieure.

Quand le taux d'occupation mémoire augmente, l'efficacité de la détection par parité se réduit et donc les erreurs détectable à la fois par la parité et l'analyse de signature sont détectées par l'analyse de signature.

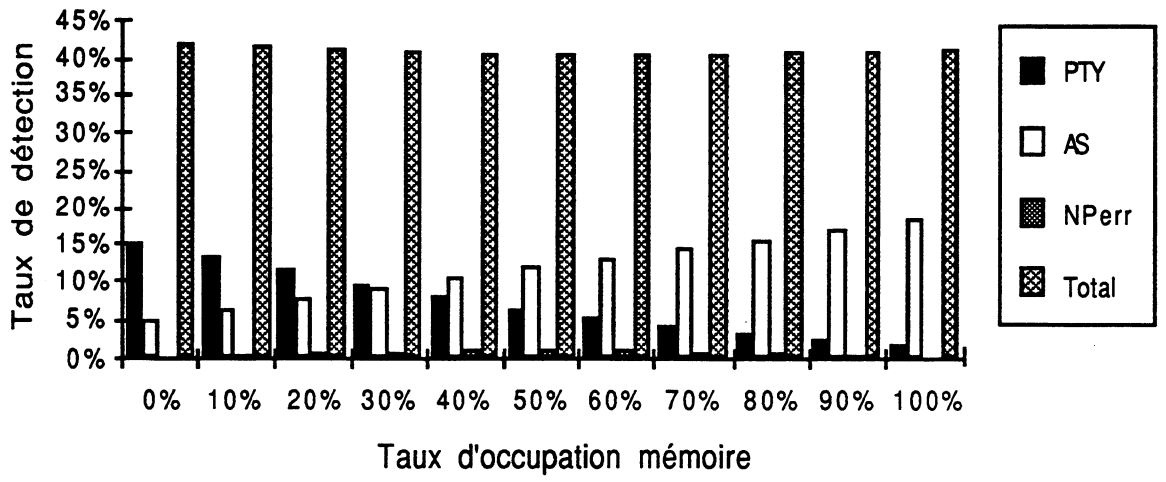


Figure 4.14 : évolution du taux de détection en fonction de la mémoire occupée.

Cette modélisation a permis d'estimer l'impact de l'analyse de signature sur le taux de détection global d'une UC. Cependant, pour obtenir des résultats plus conformes à la réalité, des expériences d'injections de fautes ont été effectuées, après réalisation du prototype. Outre la validation des dispositifs implantés, les résultats de ces expériences doivent également permettre d'affiner la modélisation effectuée grâce à la similitude entre erreurs injectées et erreurs modélisées.

4.5.2. Description des expériences d'injection de fautes

Le principe, les moyens et la mise en œuvre de l'injection de faute sur un système ne sont pas l'objet des travaux effectués dans cette thèse. Seule son utilisation en tant qu'outil de mesure sera donc abordée. Une présentation plus détaillée de l'utilisation de l'injection de fautes peut être trouvée dans [Arla 89], [Arla 90a], [Arla 90b].

Différentes méthodes d'injection de fautes ont été utilisées et présentées dans la littérature :

- forçage de valeurs (temporaires ou permanentes) sur les signaux d'entrées d'un composant [Schu 86], [Schm 82], ou sur les entrées et sur les sorties d'un composant [Arla 90a],
- coupures d'alimentation d'un circuit intégré [Karl 91],
- bombardement d'un composant avec des ions lourds [Gunn 89],
- perversion du contenu d'une mémoire, par logiciel [Chil 89], ou par matériel [Schw 87].

Actuellement, il n'existe qu'un seul appareil d'injection de faute disponible commercialement. Il s'agit du DEF Injector, conçu par l'INRS [Gera 86]. C'est cet appareil qui a été retenu pour réaliser les expériences d'injection de fautes sur le prototype d'UC réalisé.

4.5.2.1. Description de l'injecteur de fautes DEF Injector

Le mode d'injection des fautes avec le DEF Injector consiste à pervertir la mémoire contenant le code d'un programme exécuté par un microprocesseur. L'injection des fautes se fait en substituant la PROM contenant le code par une mémoire d'émulation dans laquelle des fautes ont été introduites. L'instant d'apparition de la faute est programmable ainsi que la latence de détection maximale.

Les différents modèles de fautes pouvant être programmés sur DEF Injector sont les suivant :

- Erreurs mémoire, à savoir modification permanente ou temporaire du contenu d'une adresse :
  - -> Test d'évaluation : 1 défaut/octet,
  - -> Défauts mémoire simples : 8 défauts/octet,
  - -> Défauts mémoire multiples : 255 défauts/octet.
- Défauts microprocesseur, à savoir modification de toutes les adresses contenant la même valeur (simulation d'une erreur de décodage d'instruction) :
  - -> Défauts microprocesseur simple : 8 défauts/code,
  - -> Défauts microprocesseur multiples : 255 défauts/code.
- Défauts bus, à savoir collage à 0 ou à 1 de certains signaux des bus d'adresse ou donnée de la mémoire code :
  - -> Défauts bus d'adresse simple : 8 défauts,
  - -> Défauts bus d'adresse multiples : 255 défauts,
  - -> Défauts bus d'adresse simple : 14 défauts,
  - -> Défauts bus d'adresse multiples : 16384 défauts.

L'appareil DEF Injector permet d'obtenir deux informations pour chaque expérience :

- un taux de couverture (TC) : c'est le rapport entre le nombre d'erreurs détectées et le nombre d'erreurs injectées,
- un taux de dysfonctionnement (TD) : c'est le rapport entre le nombre d'erreurs non détectées ayant entraîné un fonctionnement anormal du système (événement redouté) et le nombre d'erreurs injectées. Le comportement normal du système est défini par une procédure dite "d'activation" qui permet de savoir si l'appareil réagit correctement à un événement extérieur en cas d'erreur injectée mais non détectée.

Le déroulement d'un test est le suivant (Figure 4.15) :

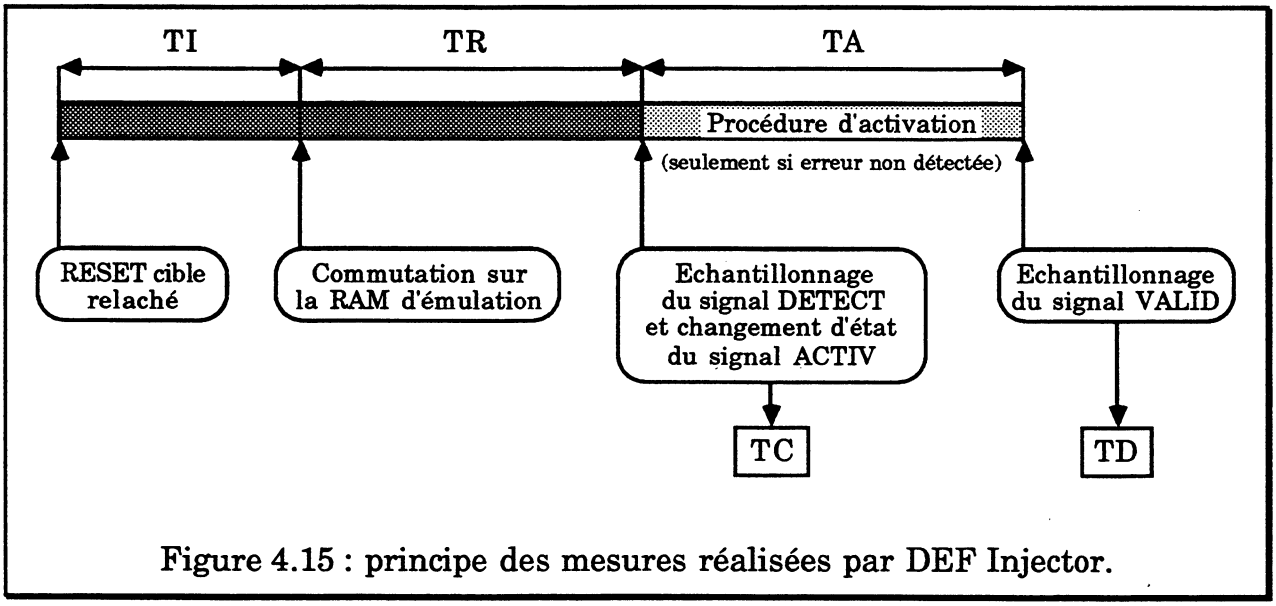


Figure 4.15 : principe des mesures réalisées par DEF Injector.

- à l'instant T=0, le RESET du système sous test est relaché par le DEF Injector,
- à l'instant TI, la mémoire programme (PROM) est remplacée par la mémoire d'émulation du DEF Injector contenant une ou plusieurs erreurs,
- à l'instant TI+TR, le signal de détection d'erreur "DETECT" est échantillonné :
  - -> si l'erreur est détectée : elle est comptabilisée dans le taux de couverture,
  - -> si l'erreur n'est pas détectée, le système sous test est "activé" par une procédure commandée par le DEF Injector sur un changement d'état du signal "ACTIV".

- à l'instant TI+TR+TA, le comportement du système est observé par le signal de réponse à l'activation "VALID" :
  -> si ce signal a changé d'état suite à l'activation, le comportement du système est considéré comme normal,
  -> si ce signal n'a pas changé suite à l'activation, le système est considéré comme défaillant, et l'erreur est comptabilisée dans le taux de dysfonctionnement.

Lors de mesures avec un test d'évaluation, les adresses accédées dans la mémoire programme pendant l'intervalle de temps [TI, TI+TR] sont enregistrées lors de tests à blanc du système (sans injection de fautes). Le contenu de ces adresses est ensuite modifié dans la mémoire d'émulation du DEF Injector conformément au modèle d'erreur (1 défaut par octet).

Lors d'un test avec procédure d'activation, des erreurs sont également injectées dans les adresses accédées uniquement pendant la phase d'activation, c'est à dire pendant l'intervalle de temps [TI+TR, TI+TR+TA]. Parmi les erreurs ainsi injectées, celles qui sont détectées avant la fin du test (TI+TR+TA) ne sont pas comptabilisées dans le taux de détection qui ne porte que sur les erreurs injectées et détectées pendant l'intervalle de temps [TI, TI+TR].

Pour un système donné, des valeurs très différentes de taux de détection et de dysfonctionnement peuvent être obtenues suivant les conditions de mesures.

Pour la mesure du taux de détection, deux paramètres sont sensibles :
- l'instant d'apparition du défaut (TI),
- la latence maximale de détection (TR).

Pour la mesure du taux de dysfonctionnement, la définition et le contrôle du comportement normal du système influencent directement la mesure. Le taux de dysfonctionnement obtenu par DEF Injector n'est donc que la probabilité d'apparition d'un événement redouté et ce taux peut varier considérablement suivant la définition de cet événement. En particulier, suivant le choix de la procédure d'activation, il est possible d'orienter ce chiffre vers une mesure de la disponibilité d'un système ou bien vers une mesure de la sécurité de ce système.

Les mesures obtenues avec l'appareil DEF Injector sont donc valables pour un ensemble complet (matériel + application exécutée + événement redouté) et ne peuvent pas servir à caractériser, de façon absolue, un matériel seul tel qu'une UC.

4.5.2.2. Déroulement des mesures

L'application exécutée par l'UC durant les expériences d'injection de fautes est un programme très classique de contrôle de chaudière à gaz. Le rôle de l'automate

consiste à faire démarrer la chaudière et ensuite à contrôler des événements sur les conditions de fonctionnement de cette chaudière (vérification de présence d'air et de combustible, détection de perte de flamme).

Cette application est de très petite taille par rapport aux capacités de stockage et de traitement de l'UC utilisée. Ceci permet de limiter le temps nécessaire aux injections de fautes mais par contre, les résultats obtenus par ces mesures ne sont pas très favorables à la mise en valeur du pouvoir de détection des mécanismes d'analyse de signature implantés. En effet, une erreur sur le séquencement de l'application a toutes les chances de faire sortir le processeur BP de la zone contenant le code. Or ce type d'erreur n'est pas détectable a priori par l'analyse de signature sur le code BP. La détection interviendra dans tous les cas soit sur erreur de parité si le processeur va dans une zone non initialisée, soit sur NPcall erreur lorsque le processeur exécutera un code de NPcall inexistant, soit sur overrun de la tâche si le processeur entre dans une boucle infinie. A la différence, sur une application de grande taille, la probabilité que le processeur BP reste dans son code suite à une erreur de séquencement est beaucoup plus forte et ce type d'erreur est détectable essentiellement par l'analyse de signature.

L'appareil d'injection de fautes nécessite le raccordement de 4 signaux avec l'automate (Figure 4.16), en plus du placement de la sonde. Certains signaux sont connectés au bus de fond de panier de l'automate et d'autres sont connectés à des entrées/sorties de l'automate par des modules TOR (Tout Ou Rien).

Ces quatre signaux sont :

- RESET : (DEFI -> TSX) utilisé pour redémarrer le système après chaque erreur injectée ; ce signal est connecté pendant les mesures au signal de reset général de l'automate du bus de fond de panier de l'automate.
- DETECT : (TSX -> DEFI) utilisé comme signal indiquant la détection d'une erreur ; ce signal est connecté pendant les mesures au signal de chien de garde général de l'automate du bus de fond de panier de l'automate.
- ACTIV : (DEFI -> TSX) utilisé lors des mesures avec procédure d'activation pour "activer" l'automate en cas d'erreur injectée mais non détectée au bout du temps de réaction programmé; ce signal est relié à une entrée du module TOR par l'intermédiaire de l'accessoire RES 50, livré avec l'appareil DEFI, utilisé en relais sur une alimentation 24V.
- VALID : (TSX -> DEFI) utilisé lors des mesures avec procédure d'activation ; son changement d'état suite à une activation (signal ACTIV) indique

un fonctionnement correct de l'automate. Ce signal est connecté directement à une sortie du module TOR (signal en logique négative, compatible avec l'entrée du DEF Injector).

L'appareil DEF Injector possède une liaison série permettant de connecter une imprimante pour obtenir des listings des expériences. Dans les mesures effectuées ici, la liaison série du DEF Injector a été reliée à un IBM PS pour que les résultats soient sur fichiers et non pas sur des listings.

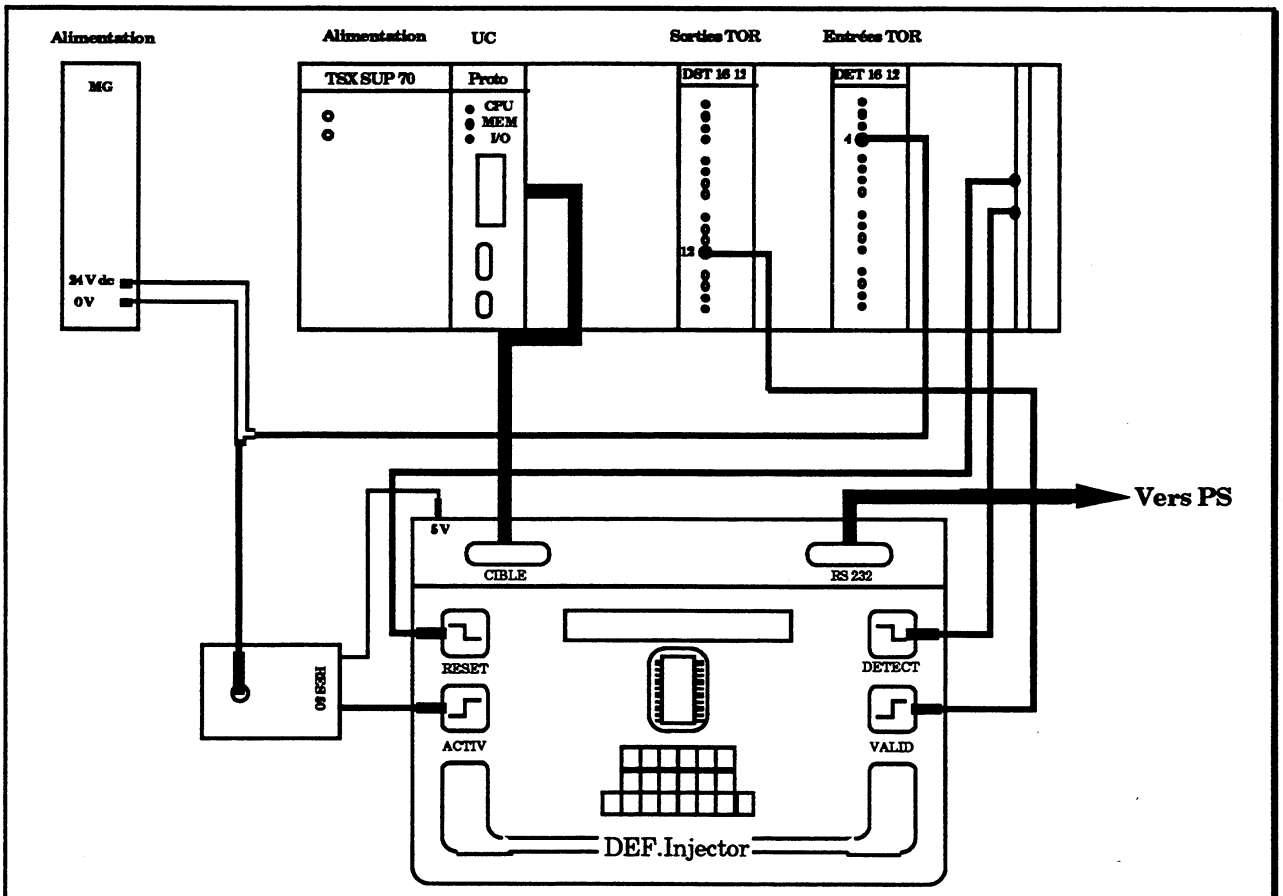


Figure 4.16 : connexion du DEF Injector sur le prototype.

Les paramètres de programmation utilisés pendant les mesures sont les suivants :

- Temps d'injection : 750 ms. Ceci correspond au moment où l'automate commence l'exécution de l'application ; le signal utilisé pour détection d'erreurs est alors inactif.
- Temps de réaction (sans time-out) : 550 ms. Ce temps est supérieur à 8 fois le cycle de tâche MAST (latence overrun) mais il est inférieur au temps de déclenchement du TO52 (compte tenu de la fréquence de fonctionnement du prototype).

- Temps de réaction (avec time-out) : 1600 ms. Ce temps est supérieur au temps de déclenchement du TO52.
- Temps d'activation : 400 ms. Ce temps est déterminé en fonction de caractéristiques propres à l'application.

La majeure partie des mesures a été faite en test d'évaluation et avec une procédure d'activation.

La procédure d'activation définit l'événement redouté dont l'apparition sera comptabilisée dans le taux de dysfonctionnement. Dans les expériences réalisées, l'activation de l'automate consiste à positionner l'entrée "présence flamme", ce qui a pour effet normalement attendu de faire passer les sorties de l'automate dans l'état correspondant à une chaudière allumée et en régime permanent en fin de procédure d'activation.

Le comportement normal de l'automate en l'absence d'erreur, compte tenu de l'application et de l'activation, consiste donc à faire démarrer la chaudière à gaz. L'événement redouté considéré dans les expériences réalisées ici est le non démarrage de la chaudière. Il faut donc remarquer que le taux de dysfonctionnement obtenu avec cette procédure d'activation donne une indication sur la disponibilité (et non la sécurité) de l'UC en cas d'erreur non détectée.

4.5.3. Injections de fautes sur le code application

Les injections de fautes sur le code BP ont été faites en plaçant la sonde de l'appareil d'injection de fautes sur les supports des PROM de la cartouche contenant le programme d'application. Différents modèles d'erreur ont été injectés (erreurs simples et multiples).

Le code BP est un code "aligné" sur les frontières de mots (16 bits), et il faut donc injecter des fautes successivement sur l'octet de poids faible et sur l'octet de poids fort (Figure 4.17).

Le format des instructions de BP se décompose en différents champs suivant le type de l'instruction (Figure 4.17). Ce type est défini par un champ des poids forts d'une instruction (code opératoire). Le reste de l'instruction est une liste de paramètres.

La plupart des instructions du jeu BP tiennent sur un seul mot mémoire à l'exception de certaines qui peuvent avoir des paramètres sur plusieurs mots et de ce fait modifient la signification des mots mémoire situés après le code opératoire.

Une erreur dans le code BP affectant une instruction comportant des paramètres et modifiant le code opératoire aura donc des conséquences plus importantes car la liste des paramètres sera interprétée d'une manière différente.

Par exemple, des paramètres pourront être transformés en codes opératoires se traduisant par l'exécution d'instructions aléatoires.

De même, une erreur transformant un code quelconque en une instruction d'un de ces deux types transformera un certain nombre de codes opératoires en paramètres et les instructions ainsi transformées seront "sautées".

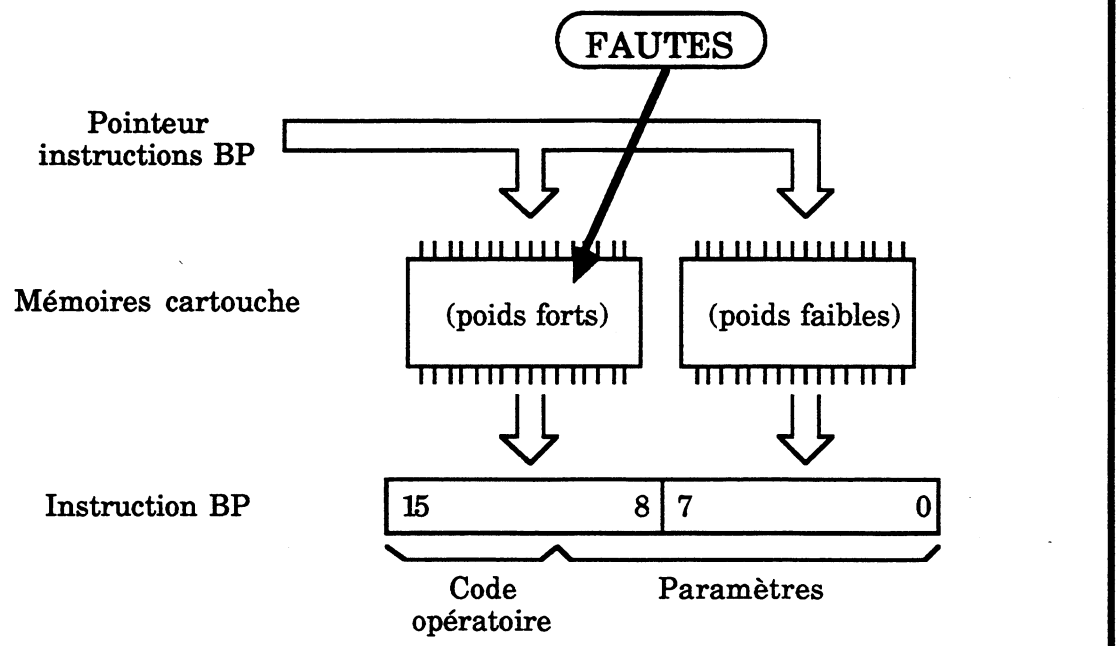


Figure 4.17 : injection de fautes sur le code BP.

Dans les deux cas, le comportement du système sera différent suivant que le séquencement du code BP est affecté ou non, et si des NPcall intempestifs sont envoyés au processeur NP :

- Si le séquencement du code BP n'est pas touché, la détection de telles erreurs est assurée par l'analyse de signature sur le code BP, par le fait que les paramètres sont signés différemment des codes opératoires,
- Si le séquencement est touché, la détection repose sur les mécanismes de détection des erreurs de séquencement BP (analyse de signature, overrun, NPcall erreur, parité).

Des NPcalls intempestifs peuvent être envoyés au processeur NP dans les conditions suivantes :

- erreur sur code opératoire transformant une instruction en NPcall,
- exécution de paramètres comme codes opératoires avec un paramètre correspondant à un code de NPcall,

- exécution de code aléatoire suite à un déséquencement BP hors du code (instructions aléatoires).

En cas de NPcall intempestif, si le code du NPcall n'existe pas, le système sera arrêté en défaut mémoire (NPcall erreur).

Certains cas d'erreur sur les NPcalls se traduisent par une indirection aléatoire du 80x86. La détection de telles erreurs ne repose plus alors sur les moyens de détection des erreurs de séquencement du code BP (quasiment infaillibles avec l'analyse de signature combinée aux moyens existants) mais sur ceux détectant les erreurs de séquencement du système (chien de garde temporel du système en particulier). Ces erreurs de séquencement 80x86 particulières sont d'autant plus difficiles à détecter qu'elles ont pour destination une zone contenant du code exécutable 8086.

Le rôle des injections de fautes sur le code BP est de chiffrer un éventuel masquage de l'analyse de signature : quel que soit le modèle des erreurs injectées, ce type d'erreur est théoriquement détecté avec la probabilité de masquage du dispositif de compaction, c'est à dire 2^{-16} . Cependant, pour évaluer l'apport de l'analyse de signature en terme de sûreté, des injections de fautes sur le code BP ont également été faites sans analyse de signature.

Dans la section suivante, les notations suivantes sont employées :
- UCAS : prototype d'UC avec analyse de signature,
- UCRF : prototype d'UC de référence (sans analyse de signature).

Des erreurs simples et des erreurs multiples ont été injectées sur le code BP. Les erreurs simples ont toutes été injectées avec la parité inhibée (/PTY) car sinon ce mécanisme aurait intercepté toutes les erreurs et les mesures ainsi réalisées n'auraient apporté aucun renseignement utile.

Des expériences ont été faites avec ou sans chien de garde temporel (TO) sur une UCRF mais par contre, les expériences sur UCAS et temps de mesure supérieur à la latence élevée n'ont pas pu être faites à cause de divers problèmes techniques liés à l'injecteur de fautes.

Pour obtenir les chiffres correspondant à une UCAS avec TO, une corrélation de différents listings a été faite. Les résultats ont été comparés erreur par erreur et non pas statistiquement et donc la confiance dans ces chiffres peut être totale, d'autant plus que le nombre d'erreurs mises en jeu dans ces comparaisons est faible.

Les chiffres ainsi obtenus sont alors indiqués en italique étant donné qu'il ne s'agit pas de résultats de mesure effectifs.

4.5.3.1. Résultats des mesures

Des erreurs simples ont été injectées sur les poids faibles et sur les poids forts du code BP pour connaître la différence au niveau de la détection pour des erreurs affectant ou non le code opératoire d'une instruction.

Le modèle de fautes injectées correspond à 1 défaut permanent par octet (test d'évaluation). Seuls des défauts permanents ont été injectés car, vis à vis de l'analyse de signature, ceci n'a qu'une incidence très limitée, et les possibilités de masquage liées à la méthode DJAM sont plus probables, à cause des erreurs permanentes dans les boucles du programme (cf 3.4.1).

Les résultats de ces expériences sont présentés sur la figure 4.18 . Pour chaque mesure, 444 erreurs ont été injectées.

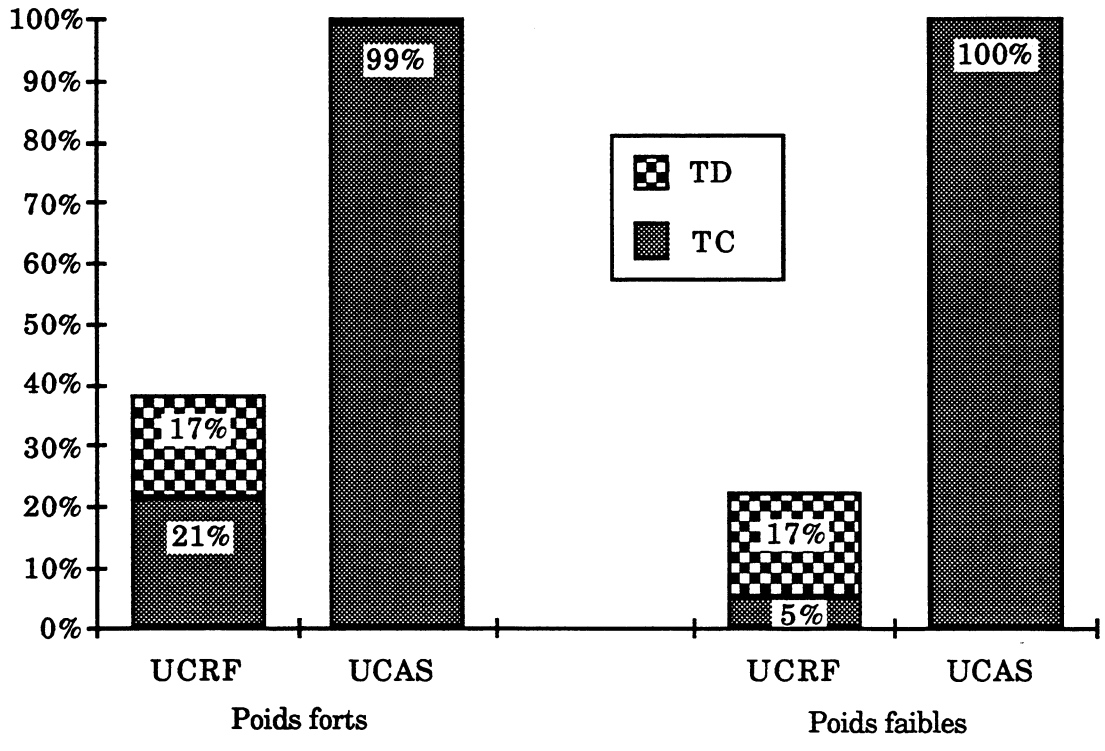


Figure 4.18 : injection d'erreurs simples sur le code application.
(TI = 750 ms, TR = 1600 ms (TO), TA = 400 ms)

Des erreurs multiples ont également été injectées, mais seulement sur les poids forts, et avec contrôle de la parité. Le modèle utilisé correspond à des défauts microprocesseur permanents multiples (255 valeurs/code). Ce modèle a été retenu car il permet d'injecter un moins grand nombre d'erreurs que le modèle de défauts mémoire multiples, tout en permettant une injection de défauts multiples.

Les résultats de ces expériences sont présentés dans la figure 4.19 . Pour chaque mesure, environ 8500 erreurs ont été injectées.

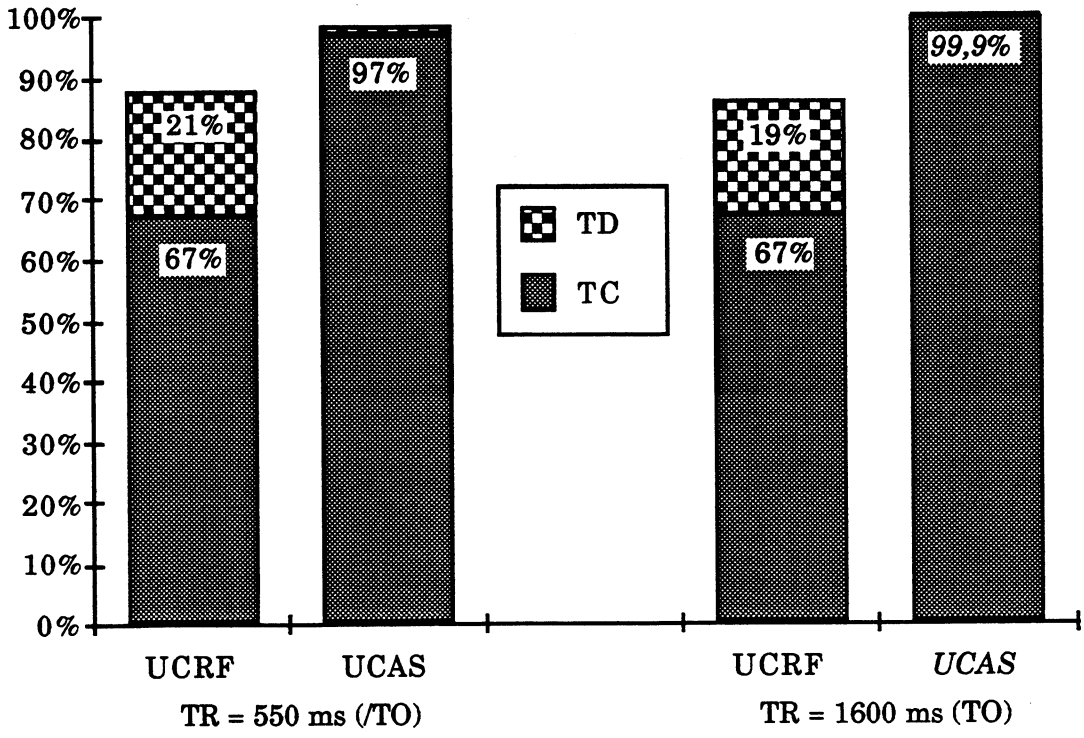


Figure 4.19 : injection d'erreurs multiples sur le code application.
(TI = 750 ms, TA = 400 ms)

4.5.3.2. Interprétation

a/ Erreurs sur les poids faibles

Le taux de détection "naturel" de l'UC envers ce type d'erreur est particulièrement faible (5%). En effet, les seuls cas d'erreurs sur les poids faibles du code BP pouvant être détectés par l'UC sont les cas où le numéro d'un NPcall est modifié. La probabilité de détection est alors égale à la proportion de NPcall erreur, plus la probabilité d'un déséquencement hors du code BP par l'exécution d'un NPcall de séquencement intempestif.

Le taux de détection sur le code est donc égal à cette probabilité de détection multipliée par la densité de NPcalls dans le code BP.

Le taux de dysfonctionnement est très fort (TD = 17%) pour ce type d'erreurs quand il n'existe que les moyens naturels de détection de l'UC (parité dévalidée).

Avec l'analyse de signature sur le code BP, le taux de détection passe à 100% : les erreurs injectées sur les poids faibles n'entraînent pas de modification des codes opératoires et leur détection par l'analyse de signature est donc assurée avec une

probabilité de masquage théorique égale à 2^{-16}. Aucun cas de masquage n'a été mis en évidence par les expériences en raison du faible nombre d'erreurs injectées.

Compte tenu du temps disponible, seules des erreurs simples ont été injectées sur les poids faibles. Ceci n'est pas gênant compte tenu du fait que la détection de ce type d'erreurs est relativement facile à prédire. Les chiffres obtenus avec des erreurs multiples ne permettraient pas de faire plus de conclusions que celles faites avec les resultats obtenus pour des erreurs simples.

b/ Erreurs sur les poids forts

Les erreurs sur les poids forts, contrairement aux erreurs sur les poids faibles, peuvent modifier le code opératoire des instructions BP et donc le comportement en cas d'erreur est moins prévisible.

Le taux de détection "naturel" de l'UC est nettement plus fort sur ce type d'erreurs :

- TC = 21 % sur erreurs simples, mesures effectuées sans parité,
- TC = 67 % sur erreurs multiples, mesures effectuées avec parité, responsable à elle seule de 50% (en valeur absolue) du taux de détection, soit seulement 17% de taux de détection en plus des 50% d'erreurs détectées immédiatement par la parité.

Le taux de dysfonctionnement, sur les erreurs de poids forts est quasiment identique aux valeurs obtenues sur les poids faibles, et ce malgré l'accroissement du taux de détection. Les erreurs de poids forts sont donc plus perturbatrices que les erreurs sur les poids faibles, ce qui parait assez normal.

Avec l'analyse de signature sur le code BP, le taux de détection de l'UC n'est pas de 100% contrairement à ce que l'on serait en droit d'attendre, et comme c'est le cas sur les poids faibles.

Pour les erreurs simples, toutes les erreurs non détectées sont comptabilisées dans le taux de dysfonctionnement (erreurs perturbatrices). Ces cas de masquage (seulement 3 cas sur 444) ont été identifiés et reproduits un par un pour déterminer la cause de ce masquage. L'analyse de ces erreurs a révélé que leur non détection n'était pas due à un masquage de l'analyse de signature mais à un masquage du dispositif censé détecter les cas d'erreurs où un test de signature n'a jamais lieu. Ces cas se produisent tous suite à l'exécution d'un NPcall avec indirection aléatoire du 80386 dans la zone de code système.

Pour les erreurs multiples, un beaucoup plus grand nombre d'erreurs (en valeurs absolue et relative) ne sont pas détectées (TC = 97%) à latence courte. Par ailleurs, toutes les erreurs non détectées ne se retrouvent pas comptabilisées dans le

taux de dysfonctionnement, comme c'est le cas pour les erreurs simples. Ce phénomène est dû à des erreurs qui entraînent un déséquencement majeur de BP dans une tâche auxiliaire, phénomène dont une partie ne peut être détecté que par overrun, donc à latence élevée.

La corrélation de différents listings a permis d'obtenir le taux de détection et le taux de dysfonctionnement à latence élevée en présence de l'analyse de signature, et dans ce cas, seulement 3 erreurs sur 8670 ne sont pas détectées et toutes se retrouvent au niveau du taux de dysfonctionnement. L'analyse de ces erreurs a montré que la non détection de ces erreurs était due à un phénomène identique à celui du masquage constaté sur les erreurs simples.

4.5.4. Injections de fautes sur le code système

Les injections de fautes sur le code système ont été faites en plaçant la sonde de l'injecteur de fautes à la place d'une des deux PROM contenant le système. Les injections n'ont été faites que sur les poids forts des mots mémoire. En effet, le code 8086 n'est pas aligné sur les frontières des mots (16 bits) et le format des instructions est très diversifié (de 1 à 6 octets). Statistiquement, la probabilité d'injecter des erreurs sur un code opératoire ou sur une adresse ou une donnée immédiate ne dépend pas du fait que les erreurs soient injectées sur les poids faibles ou sur les poids forts des mots mémoires correspondant au code 8086. Pour diminuer la durée des tests, les injections sur les poids faibles n'ont pas eu lieu, sauf quelques unes, pour vérifier que les chiffres obtenus étaient tout à fait similaires à ceux obtenus sur les poids forts.

Compte tenu du nombre important d'erreurs à injecter dans le système, seul le modèle 1 défaut par octet a pu être utilisé. Le modèle 8 défauts par octet aurait donné des résultats quasiment identiques et le modèle 255 défauts par octet était impraticable vu le nombre d'erreurs phénoménal qu'il aurait fallu injecter.

Quant à la nature des erreurs injectées, le modèle d'erreurs permanentes a été utilisé pour l'ensemble des mesures, mis à part quelques essais. La raison de ce choix vient du fait qu'il n'existe pas de moyen spécifique de détection des erreurs permanentes sur le code système, c'est à dire que la détection des erreurs, qu'elles soient transitoires ou permanentes, repose sur les mêmes mécanismes. Par contre, en cas d'erreurs transitoires, le nombre d'erreurs réellement injectées au niveau du processeur est beaucoup plus faible, car une partie des erreurs transitoires injectées dans la mémoire n'arrivent pas jusqu'au microprocesseur (erreurs injectées dans la file d'attente du 80386 avant une rupture de séquence en particulier). Les chiffres obtenus avec des erreurs transitoires sont donc différents de ceux obtenus avec des erreurs permanentes : le taux de détection et le taux de

dysfonctionnement baissent sensiblement. Pour obtenir les chiffres de taux de dysfonctionnement les plus défavorables possible, le modèle d'erreurs permanentes a été retenu. Ces chiffres de taux de dysfonctionnement peuvent être pris comme des pires cas pour les erreurs transitoires.

Par ailleurs, le but de ces injections est de faire des comparaisons pour chiffrer les taux de détection relatifs des différents mécanismes de test en ligne, et en particulier l'amélioration apportée par l'analyse de signature. Dans la mesure ou les défauts injectés le sont toujours sur un seul octet du code à la fois, le modèle de faute (transitoire ou permanente, simple ou multiple) n'a quasiment aucune incidence sur le taux de détection de la vérification de flot de contrôle au niveau du système, à la différence de certains autres mécanismes (time-out, overrun, débordements de pile ...). L'amélioration révélée par les mesures avec erreurs permanentes est donc là aussi un pire cas.

Les expériences d'injection de fautes sur le code système ont été effectuées de manière à pouvoir chiffrer le taux de détection effectif des différents mécanismes présents sur le prototype d'UC. Quatre mécanismes ont ainsi été isolés en comparant les résultats de mesures sur une UC disposant du mécanisme et une UC n'en disposant pas :

- Analyse de signature (AS),
- Parité (PTY),
- Chien de garde temporel (TO),
- Exception du 80386 en mode réel (Privilèges 386)

Pour cela, les mesures ont été effectuées sur trois types d'UC :

- UC basée sur un 80386, avec analyse de signature (UCAS),
- UC basée sur un 80386 sans analyse de signature (UCRF),
- UC basée sur un 8086 sans analyse de signature (UC86).

Pour chaque type d'UC, des mesures ont été faites en validant ou non certains mécanismes de test :

- avec parité (PTY),
- sans parité (/PTY),
- avec chien de garde temporel (TO), TR = 1,6 s,
- sans chien de garde temporel (/TO), TR = 550 ms.

4.5.4.1. Tableaux récapitulatifs des résultats de mesures

Les résultats présentés ici sont des moyennes obtenues sur l'ensemble des fautes injectées. Le nombre d'erreurs injectées varie souvent entre le taux de couverture et le taux de dysfonctionnement (toujours inférieur pour la mesure du

taux de couverture) à cause des erreurs injectées pendant la procédure d'activation (cf 4.5.2.1).

Tableau 4.3 : mesures à latence élevée et avec parité (TO PTY).

(TO PTY)	TC	detect/inj	TD	perturb/inj
UCAS	46,80%	(2849/6087)	2,71%	(167/6157)
UCRF	42,58%	(2590/6083)	3,27%	(201/6153)
UC86	40,05%	(2474/6178)	4,26%	(266/6249)

Tableau 4.4 : mesures à latence faible et avec parité (/TO PTY).

(/TO PTY)	TC	detect/inj	TD	perturb/inj
UCAS	44,35%	(2609/5883)	6,36%	(392/6166)
UCRF	39,35%	(2291/5822)	6,72%	(409/6086)
UC86	28,31%	(1675/5916)	15,93%	(984/6178)

Tableau 4.5 : mesures à latence élevée et sans parité (TO /PTY).

(TO /PTY)	TC	detect/inj	TD	perturb/inj
UCAS	46,07%	(2805/6088)	2,96%	(182/6158)
UCRF	41,98%	(2555/6086)	3,18%	(196/6156)
UC86	mesure non effectuée		mesure non effectuée	

Tableau 4.6 : mesures à latence faible et sans parité (/TO /PTY).

(/TO /PTY)	TC	detect/inj	TD	perturb/inj
UCAS	44,07%	(2564/5818)	5,82%	(354/6085)
UCRF	39,30%	(2310/5875)	7,19%	(442/6148)
UC86	26,20%	(1547/5905)	17,46%	(1078/6174)

Les chiffres des tableaux précédents sont utilisés directement pour extraire les chiffres présentés dans les tableaux 4.7 à 4.13 (cf 4.5.4.2 et 4.5.4.3). Pour chaque mécanisme, il est présenté la différence (en valeur absolue) entre chaque taux de couverture et taux de dysfonctionnement (respectivement Δ TC, Δ TD).

4.5.4.2. Amélioration apportée par l'analyse de signature

Les chiffres des tableaux précédents (résumés dans le tableau 4.7) mettent en évidence que l'amélioration apportée par l'analyse de signature sur le prototype d'UC est assez faible en valeur absolue pour les erreurs du système, mais cette amélioration existe, malgré la puissance de la détection des privilèges du 80386 (cf 4.5.4.3).

Tableau 4.7 : efficacité de l'analyse de signature implantée
Les colonnes "gain TC" et "dimin. TD" représentent la modification en pourcentage du TC et du TDpar rapport à une UC dépourvue d'analyse de signature.

	Δ TC	gain TC	ΔTD	dimin. TD
TO PTY	4,23	9,93%	-0,55	-16,97%
/TO PTY	5,00	12,70%	-0,36	-5,40%
TO /PTY	4,09	9,75%	-0,23	-7,73%
/TO /PTY	4,77	12,14%	-1,37	-19,76%

L'amélioration apportée par la vérification de flot de contrôle est plus sensible sur le taux de détection que sur le taux de dysfonctionnement. Ceci est en partie explicable par le fait que les erreurs système affectant le fonctionnement de l'analyseur de signature sont toutes détectées par l'analyse de signature alors qu'en l'absence d'analyse de signature ces erreurs n'ont pas de raison de perturber le système. Cependant, les erreurs dans le fonctionnement de l'analyseur ne sont pas les seules à être détectées pas l'analyse de signature, comme en témoignent les chiffres concernant le taux de dysfonctionnement.

En analysant les chiffres, on peut constater que l'amélioration du taux de couverture par l'analyse de signature est peu sensible à la présence des autres mécanismes (TO et PTY). On notera toutefois que l'amélioration est légèrement plus forte pour les mesures effectuées à latence faible, ce qui signifie que des erreurs détectables par le chien de garde temporel sont interceptées par l'analyse de signature, très certainement à cause de processus d'E/S intempestifs (erreurs dangereuses) lors d'un déséquencement du microprocesseur.

On pourra également constater que l'amélioration apportée par l'analyse de signature est plus forte quand la parité est en service, ce qui à première vue ne parait pas du tout logique. En effet, sur le prototype d'UC étudié, les seules erreurs du système détectables par parité sont un déséquencement du BP dans une zone ne contenant pas de mémoire. Ces erreurs sont détectables en l'absence de parité par overrun d'une tâche, NPcall erreur, analyse de signature, ou à travers le système si un code de NPcall provoque une indirection aléatoire du 80386. L'amélioration de l'analyse de signature en l'absence de parité devrait donc être au moins égale à l'amélioration obtenue avec parité en service, voire supérieure.

De même, les chiffres d'amélioration du taux de dysfonctionnement, en fonction des différents dispositifs de test utilisés, ne semblent pas très cohérents.

Dans les deux cas, ces incohérences sont certainement dues à l'incertitude des mesures. En particulier, le comportement d'une UC à la suite d'une erreur ayant

entraîné un déséquencement majeur d'un processeur (BP ou NP) n'est pas reproductible d'une mesure à l'autre, et donne lieu à des effets souvent différents pour une même erreur. Parmi ces effets, certains seront perturbateurs pour l'application et d'autres ne le seront pas.

Le fait que les comparaisons soient effectuées sur les résultats statistiques des injections de fautes explique en partie ce manque de cohérence. Cependant, les moyens d'observation de l'appareil d'injection de fautes n'ont pas permi de réaliser des comparaisons erreurs par erreurs, ce qui aurait donné des résultats beaucoup plus précis. Les conclusions obtenues avec ces résultats statistiques sont néanmoins très instructives.

4.5.4.3. Efficacité des mécanismes de base

Dans les tableaux 4.8 à 4.13 , la colonne "% TC" représente le pourcentage du taux de détection attribuable au mécanisme considéré, sur une UC disposant de ce mécanisme. La colonne "dimin. TD" représente la diminution relative du taux de dysfonctionnement par rapport à une UC qui ne disposerait pas de ce mécanisme.

4.5.4.3.1. Chien de garde temporel et privilèges 386

Parmi les mécanismes de base, le TO52 et les privilèges du 386 sont particulièrement efficaces, mais ils détectent globalement les mêmes erreurs.

L'efficacité des privilèges du 386 est bien visible en comparant les résultats d'une UCRF (80386) avec ceux d'une UC86 (8086), toutes les deux dépourvues d'autres mécanismes (pas de TO52, pas d'analyse de signature et pas de parité). Près de 30% du taux de détection est dû à une violation des privilèges du 386 (Tableau 4.8).

Tableau 4.8 : efficacité du 386 en mode réel.

	Δ TC	% TC	ΔTD	dimin. TD
TO PTY	2,53	5,95%	-0,99	-23,26%
/TO PTY	11,04	28,05%	-9,21	-57,81%
/TO /PTY	13,10	33,34%	-10,27	-58,82%

L'efficacité du TO52 est bien visible sur les chiffres obtenus sur l'UC86 (qui ne dispose donc pas d'un 80386). Le TO52 est responsable à lui seul de près de 30% également du taux de détection d'une UC86 (Tableau 4.9).

Tableau 4.9 : efficacité du TO52 sur une UC86.

UC86	Δ TC	% TC	ΔTD	dimin. TD
PTY	11,73	29,30%	-11,67	-73,26%

Dans les deux cas, la quantité d'erreurs détectées se retrouve presque en totalité amputée sur le nombre d'erreurs perturbatrices, preuve que les erreurs détectées par ces mécanismes sont particulièrement dangereuses.

L'explication la plus logique de ces résultats est que les erreurs détectées par ces deux mécanismes sont des erreurs de séquencement du NP. En présence du 386, ces erreurs de séquencement se traduisent le plus souvent par une sortie du 386 de l'espace d'adressage du 86 (rapport des tailles des zones d'adressage : 1/1000). En l'absence de 386, par contre, les déséquencements majeurs du NP se traduisent, dans le meilleur des cas, par un accès à une zone de parité non initialisée, ou par une boucle infinie stoppée par TO52.

Les erreurs d'adressage de NP sur des données peuvent donner lieu à des cas de détection tout à fait similaires vis à vis de la parité et des privilèges 386, mais dans une proportion a priori beaucoup plus faible que pour les erreurs de séquencement, compte tenu de la très forte corrélation entre erreurs détectées et erreurs perturbatrices (les erreurs sur les données sont en moyenne moins perturbatrices que les erreurs de séquencement).

La différence sur le taux de détection global entre une UC86 et une UCRF peut s'expliquer par le fait que certains déséquencements du 8086 se terminent par un retour plus ou moins heureux du NP dans une zone de code 86, ce qui est quasiment impossible dans le cas d'un 80386 (sauf si un NPcall est exécuté avec une indirection aléatoire).

Les erreurs détectées par le TO52 sur une UCRF et sur une UCAS sont donc soit des boucles infinies du 386 dans l'espace d'adressage du 86 (peu probable), soit un "oubli" du NP de revalider les interruptions de l'horloge temps réel (plus probable).

Là aussi la relation est directe entre erreurs détectées et erreurs perturbatrices (Tableau 4.10).

Tableau 4.10 : efficacité du TO52 sur une UC basée sur un 386.

UC (386)	Δ TC	% TC	ΔTD	dimin. TD
UCAS (PTY)	2,46	5,25%	-3,65	-57,34%
UCAS (/PTY)	2,00	4,35%	-2,86	-49,20%
UCRF (PTY)	3,23	7,58%	-3,45	-51,39%
UCRF (/PTY)	2,68	6,39%	-4,01	-55,71%
moyenne	2,59	5,89%	-3,49	-53,41%

4.5.4.3.2. Parité sur UCRF et UCAS (80386)

Les seules erreurs détectables par la parité sur une UCRF ou UCAS sont les déséquencements de BP dans une zone de parité non initialisée (absence de mémoire). Il est donc normal de constater la très faible efficacité de la parité sur une UC basée sur un 80386, aussi bien avec que sans analyse de signature (respectivement tableau 4.11 et tableau 4.12).

Tableau 4.11 : efficacité de la parité sur une UCAS.

UCAS	Δ TC	% TC	ΔTD	dimin. TD
TO	0,73	1,56%	-0,24	-8,23%
/TO	0,28	0,63%	+0,54	+9,28%

Tableau 4.12 : efficacité de la parité sur une UCRF.

UCRF	Δ TC	% TC	ΔTD	dimin. TD
TO	0,60	1,40%	+0,09	+2,60%
/TO	0,05	0,13%	-0,47	-6,52%

On constatera également que l'efficacité de la parité est plus forte sur les mesures effectuées à latence élevée, ce qui parait logique. En effet, un déséquencement BP peut donner lieu à une exécution aléatoire du BP plus ou moins longue avant que celui-ci n'accède une zone ne contenant pas de mémoire. Le taux de détection de la parité est donc proportionnel à la latence, mais cette latence est bornée par celle de l'overrun de la tâche déséquencée.

Il est par contre surprenant de constater que l'efficacité de la parité est plus forte lorsque l'analyse de signature est présente, ce qui est peu logique : les chiffres semblent indiquer que, pour la parité et l'analyse de signature, le pouvoir de détection individuel est meilleur lorsque les deux mécanismes sont présents. Là encore, ces incohérences sont certainement dues à l'incertitude des mesures.

En ce qui concerne les chiffres sur le taux de dysfonctionnement, ils ne sont pas très cohérents, eux non plus, et comme pour l'analyse de signature, ceci est certainement dû à la non reproductibilité du comportement de l'UC en cas de déséquencement d'un des deux processeurs (essentiellement BP dans ce cas). En particulier, certains résultats mettent en évidence une augmentation du taux de dysfonctionnement (signes "+" dans la colonne "dimin. TD") dans le cas où la parité est présente, ce qui n'est pas logique.

4.5.4.3.3. Parité sur UC86

L'efficacité de la parité est supérieure sur une UC86 et ceci est normal car, compte tenu de la logique de bus spécifique au 8086, la parité peut alors détecter des erreurs en lecture mémoire du 8086 (hors zone système). Des erreurs d'adressage du microprocesseur peuvent donc être détectées par parité en cas de déséquencement de celui-ci et en cas d'erreurs d'adressage de données (Tableau 4.13).

Tableau 4.13 : efficacité de la parité sur une UC86.

UC86	Δ TC	% TC	ΔTD	dimin. TD
/TO	2,11	7,47%	-1,53	-8,78%

4.5.4.4. Synthèse des résultats pour les erreurs injectées dans le système

Dans cette section, les valeurs des taux individuels pour chaque mécanisme sont obtenues à partir des résultats statistiques des mesures en faisant la différence, pour un mécanisme donné, des taux obtenus avec une UC comportant ce mécanisme et une UC ne le comportant pas, les autres mécanismes étant invalidés dans les deux cas. Ce mode de calcul a été choisi en l'absence de possibilité d'observation directe de chaque mécanisme. Ceci induit donc des risques d'erreurs d'estimation à cause de la non reproductibilité de certains phénomènes.

4.5.4.4.1. Détection des erreurs par mécanisme

Dans le cas de la parité sur les UCAS et UCRF, le taux de détection est pris à latence élevé, donc en présence du TO52, car le taux de détection de la parité augmente avec la latence, mais les erreurs détectées par parité le sont toujours avant détection par TO52.

Par exemple :

TC (AS) = TC_{UCAS} (/PTY /TO) - TC_{UCRF} (/PTY /TO)

TC (TO/UCRF) = TC_{UCRF} (/PTY TO) - TC_{UCRF7} (/PTY /TO)

TC (PTY/UCRF) = TC_{UCRF} (PTY TO) - TC_{UCRF} (/PTY TO)

TC (386) = TC_{UCRF} (/PTY /TO) - TC_{UC86} (/PTY /TO)

La comparaison de la répartition de la détection des erreurs sur chaque type d'UC est présentée sur la figure 4.20 . Sur ce graphique, on constate que la différence entre le taux de détection d'une UC (avec l'ensemble des mécanismes) et la somme des taux individuels de chaque mécanisme est une constante qui correspond au taux de couverture "naturel" d'une UC (TC_{UC86} (/PTY /TO)). Les mécanismes contribuant à ce chiffre sont appelés "autres", c'est à dire la détection par overrun, par NPcall erreur et à travers les vérifications logicielles du système.

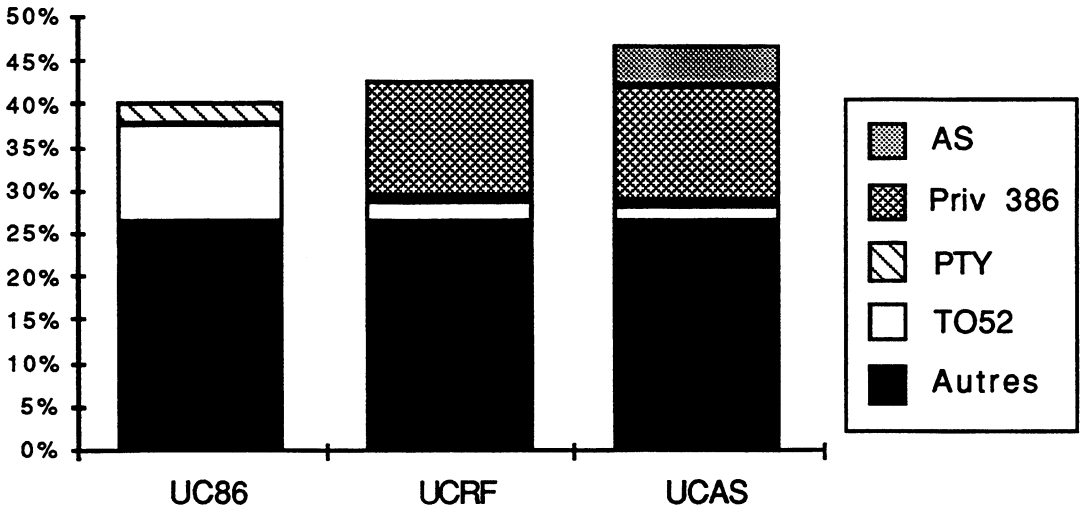


Figure 4.20 : comparaison de la répartition du taux de détection.

La figure 4.21, quant à elle, représente l'efficacité relative de chaque mécanisme, en pourcentage du taux de détection, pour les trois types d'UC.

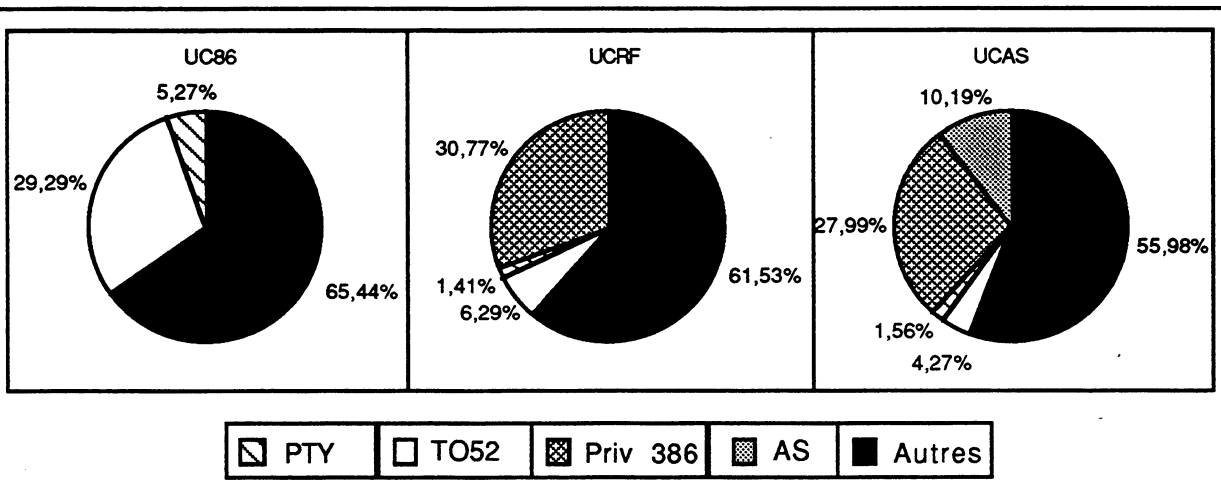


Figure 4.21 : répartition de la détection des erreurs
(en % du taux de couverture).

On constate, sur les figures 4.20 et 4.21, que l'efficacité relative de la détection des erreurs par chien de garde temporel diminue sensiblement grâce à la présence du 80386 mais également grâce à la présence de l'analyse de signature. Ces deux mécanismes possèdent donc des propriétés de détection intéressante d'un point de vue sécurité car le comportement d'une UC avant détection d'une erreur par chien de garde temporel est totalement imprévisible. La diminution du nombre d'erreurs détectées par chien de garde temporel est donc bénéfique sur le plan de la sécurité.

4.5.4.4.2. Répartition des erreurs potentiellement perturbatrices

Comme pour la synthèse sur le taux de couverture, les valeurs des taux individuels pour chaque mécanisme sont obtenus à partir des chiffres de mesures en faisant la différence, pour un mécanisme donné, des taux obtenus avec une UC comportant ce mécanisme et une UC ne le comportant pas, les autres mécanismes étant invalidés (sauf dans le cas de la parité sur les UCAS et UCRF dont les taux sont pris à latence élevée, donc en présence du TO52).

Les taux comparés ici sont par contre des taux de dysfonctionnement. Pour chaque comparaison, la différence entre deux taux de dysfonctionnement s'explique par la détection d'erreurs qui, lorsqu'elles ne sont pas détectées, conduisent à un dysfonctionnement.

Les chiffres obtenus ici sont donc des taux de couverture par rapport à des erreurs potentiellement perturbatrices (noté TC/d). Par exemple:

-TC/d (AS) = TD $_{UCAS}$ (/PTY /TO) - TD $_{UCRF}$ (/PTY /TO)
-TC/d (TO/UCRF) = TD $_{UCRF}$ (/PTY TO) - TD $_{UCRF}$ (/PTY /TO)
-TC/d (PTY/UCRF) = TD $_{UCRF}$ (PTY TO) - TD $_{UCRF}$ (/PTY TO)
-TC/d (386) = TD $_{UCRF}$ (/PTY /TO) - TD $_{UC86}$ (/PTY /TO)

La somme du taux de dysfonctionnement d'une UC et de tous les TC/d des mécanismes présents sur cette UC est une constante qui correspond au taux de dysfonctionnement "naturel" d'une UC (TD $_{UC86}$ (/PTY /TO)). Le taux de dysfonctionnement d'une UC est représenté sur les figures 4.22 et 4.23 par un damier (TD = échec de la détection).

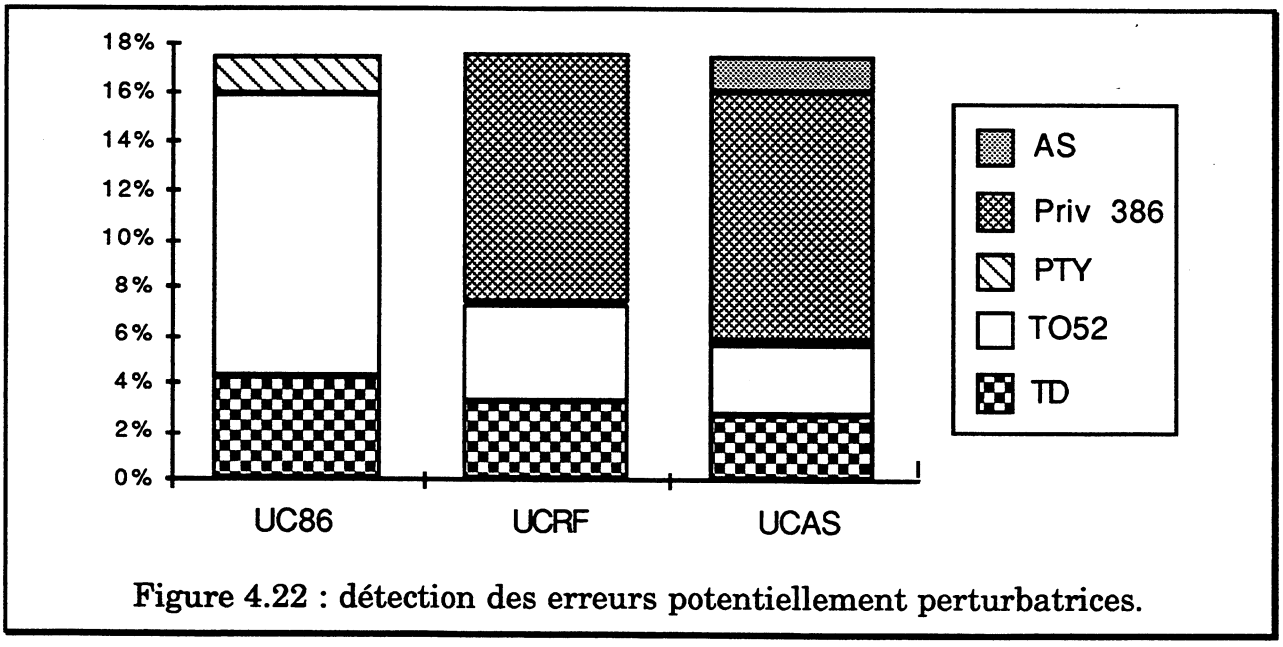


Figure 4.22 : détection des erreurs potentiellement perturbatrices.

Les figures 4.22 et 4.23 représentent la répartition des erreurs perturbatrices, après détection des moyens "naturels" de l'UC. Comme pour les résultats obtenus sur le taux de détection, on constate une diminution de l'efficacité relative du chien de garde temporel, grâce à la présence d'un 80386 et de l'analyse de signature, ce qui est bon signe d'un point de vue sécurité.

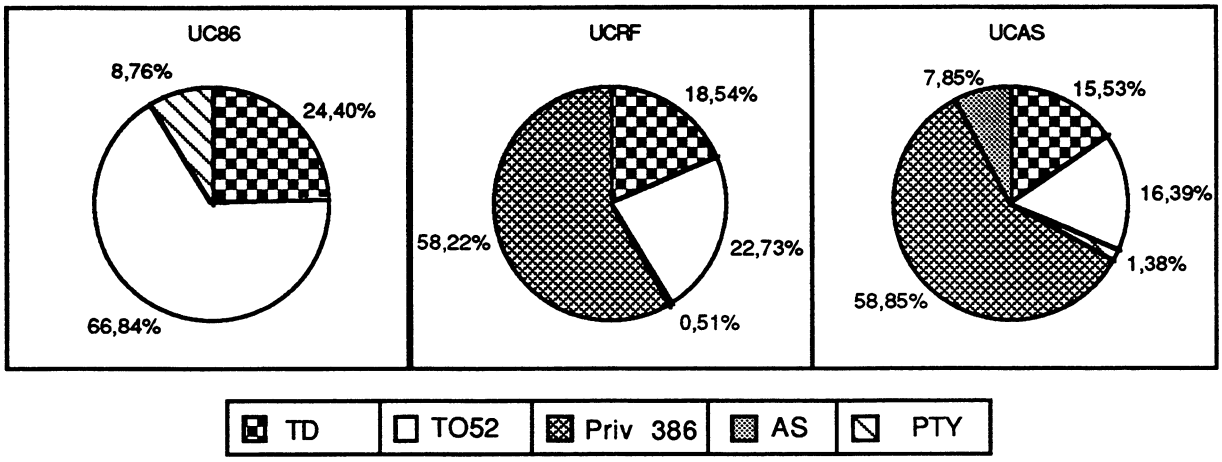


Figure 4.23 : répartition de la détection erreurs potentiellement perturbatrices.

4.5.4.5. Mesures complémentaires

4.5.4.5.1. Erreurs d'Entrées/Sorties

La détection des erreurs affectant un processus d'E/S ne fait pas réellement partie des mécanismes de base d'une UC. En effet, il faut que l'application gère elle-même les bits système positionnés en cas de détection d'une telle erreur, car ce type de vérification sert essentiellement à détecter un problème lié aux modules d'entrées/sorties et non à l'UC elle-même. Cependant, dans la mesure où des erreurs de l'UC peuvent donner lieu à ce même type d'erreur, les mesures présentées en section 4.5.4.1 incluent ce type de détection dans les mécanismes de base de l'UC (en provoquant une boucle infinie de la tâche FAST, détectée rapidement par overrun, en cas d'erreur d'E/S). Pour avoir une idée de la quantité d'erreurs de l'UC ainsi détectées, une série de mesures a été faite en supprimant la détection des erreurs d'E/S dans les mécanismes de base de l'UC (pas de contrôle des bits système). Les résultats sont indiqués dans le tableau 4.14 .

Comme on peut le remarquer sur le tableau 4.14, la détection d'erreurs du système à travers les erreurs d'E/S n'est pas négligeable et la presque totalité des erreurs détectées sont perturbatrices et représentent presque 20% du taux de dysfonctionnement.

Tableau 4.14 : efficacité de la détection par erreur d'Entrées/Sorties.

(/TO PTY)	TC	detect/inj	TD	perturb/inj
UCRF (+ES)	39,35%	(2291/5822)	6,72%	(409/6086)
UCRF - ES	37,37%	(2196/5876)	8,33%	(512/6146)
Efficacité absolue	1,98%		-1,61%	
Efficacité relative	5,29%		-19,33%	

4.5.4.5.2. Erreurs détectées par l'application

Il existe des cas d'erreurs dans lesquels l'état de l'application est modifié (dysfonctionnement dans les mesures de la section 4.5.3.1) mais où une anomalie est détectée par les vérifications faites par l'application elle-même (événements n'arrivant pas dans une plage de temps fixée, par exemple). Ces cas d'erreurs, s'ils correspondent effectivement à un problème de disponibilité de l'UC, ne sont pas pour autant dangereux car l'application est dans un état sûr suite à la détection d'une erreur. Une série de mesures a donc été faite pour quantifier ce phénomène. Pour cela, le signal de détection relié à l'entrée "DETECT" du DEF Injector n'est pas le signal de chien de garde général de fond de panier mais une sortie du module TOR indiquant l'état de l'application (signal OK).

Tableau 4.15 : efficacité de la détection par l'application sur une UCAS.

(TO PTY)	TC	detect/inj	TD	perturb/inj
UCAS	46,80%	(2849/6087)	2,71%	(167/6157)
UCAS + OK	47,19%	(2903/6152)	2,60%	(162/6222)
Efficacité absolue	0,38%		-0,11%	
Efficacité relative	0,82%		-4,01%	

Tableau 4.16 : efficacité de la détection par l'application sur une UCRF.

(TO PTY)	TC	detect/inj	TD	perturb/inj
UCRF	42,58%	(2590/60837)	3,27%	(201/6153)
UCRF + OK	42,93%	(2640/6149)	2,92%	(182/2640)
Efficacité absolue	0,36%		-0,34%	
Efficacité relative	0,84%		-10,47%	

La détection d'erreur à travers l'application n'est donc pas très efficace pour l'application considérée, en raison du peu de vérifications qui y sont effectuées. Néanmoins il existe tout de même des cas d'erreurs du système qui sont interceptés par l'application. Les chiffres des tableaux 4.15 et 4.16 indiquent également que les

erreurs perturbatrices détectées par l'application sont moins nombreuses en présence de l'analyse de signature.

4.5.5. Comparaison des mesures avec la modélisation du taux de détection

En section 4.5.1 , une modélisation du taux de détection a été présentée afin d'évaluer l'amélioration apportée par l'analyse de signature. Le modèle de fautes utilisé considérait les défauts (transitoires) sur les bus d'une UC et affectait à chaque bus une probabilité d'erreur calculée à partir des taux de défaillance des composants connectés à un bus. La comparaison directe des résultats de ce modèle avec les résultats d'injection de fautes n'est donc pas très significative.

Pour que la comparaison ait un sens, il faut remplacer les probabilités du modèle original par des valeurs représentant le modèle de fautes injectées. Ceci a été fait en affectant une probabilité égale à 1 pour l'occurrence d'erreurs sur le bus d'instruction du 80386 et une probabilité nulle pour les erreurs sur les autres bus.

Compte tenu des résultats de mesures obtenus, le modèle original a été légèrement modifié, ceci afin de prendre en compte les possibilités de détection d'erreurs d'adressage du 80386. Dans le modèle original, en effet, les possibilités de détection du 386 en mode réel n'avaient pas été prises en compte car leur efficacité était alors sous-estimée. De plus, les paramètres du modèle ont été adaptés pour être plus en rapport avec les caractéristiques de l'application exécutée (faible taille mémoire occupée en particulier).

Les résultats de modélisation ainsi obtenus sont représentés sur la figure 4.24 avec les résultats obtenus par injections de fautes sur le code système.

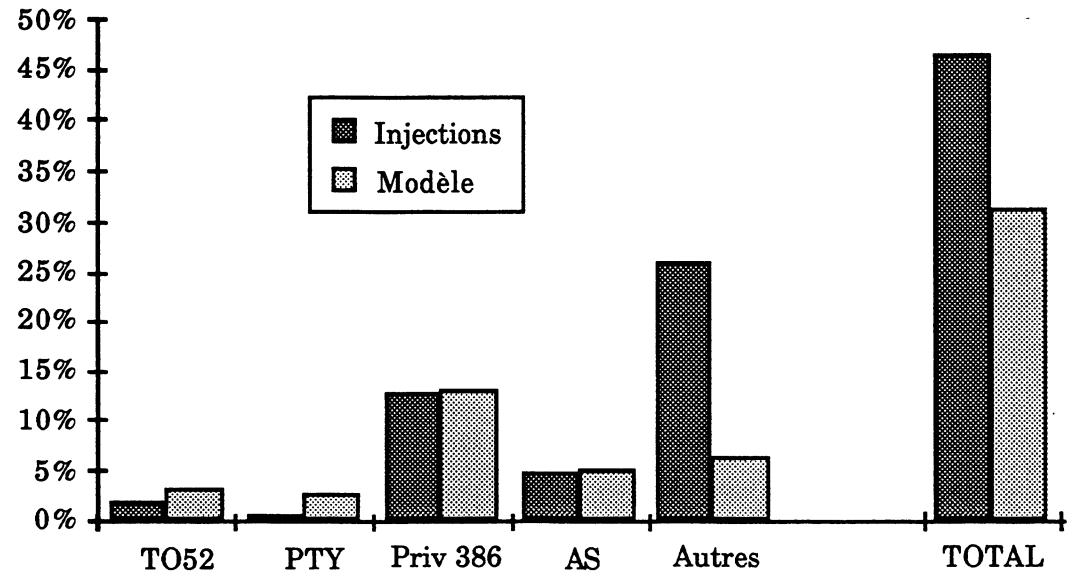


Figure 4.24 : comparaison des résultats de mesures avec la modélisation.

On pourra constater sur la figure 4.24 que les valeurs de taux de détection obtenues par modélisation pour les principaux mécanismes (386, TO, PTY, AS) ne sont pas très éloignées (en valeur absolue) des chiffres obtenus par injection de fautes. En particulier, la hiérarchie de ces mécanismes, d'un point de vue taux de détection, est identique dans les deux cas.

Par contre, le total des erreurs détectées est largement plus faible sur les résultats de modélisation. Cette différence se retrouve également au niveau du taux de détection des moyens appelés "autres", dont l'efficacité est largement sous-estimée par la modélisation. Ceci est dû au fait que des erreurs sont détectées par des moyens de détection logiciels du système, dispositifs extrêmement difficiles à prendre en compte dans une modélisation.

Comme dans les résultats de mesure, on remarquera que l'efficacité de l'analyse de signature est moins importante que celle obtenue lors de la modélisation initiale. Ceci est dû à plusieurs phénomènes :

- le modèle d'erreur n'est pas le même : les erreurs injectées dans le système ne reflètent pas l'ensemble des erreurs prise en compte dans le modèle original (erreurs sur les bus du PIU en particulier),
- les privilèges du 386 (absents dans le modèle initial) enlèvent une partie de son efficacité à l'analyse de signature telle qu'elle a été implantée,
- les paramètres du modèle propres aux conditions d'injections de fautes ne jouent pas en faveur de l'analyse de signature (taille mémoire occupée en particulier) alors que dans la modélisation originale, des valeurs moyennes avaient été utilisées.

L'accord entre le modèle "affiné" et les résultats d'injections de fautes est donc relativement bon. Des ordres de grandeurs peuvent ainsi être déduits de ce modèle pour des hypothèses de fautes plus larges que celles employées lors des injections.

Par ailleurs, cette étude de cas montre bien la difficulté d'obtention d'un modèle significatif et la nécessité de combiner des injections de fautes avec une modélisation.

4.5.6. Conclusion sur l'efficacité

L'efficacité de la vérification du flot de contrôle d'une UC a été évaluée par une modélisation du taux de détection global d'une UC et par des expériences d'injection de fautes sur le code application et sur le code système. Les résultats de ces expériences ont montré le bon pouvoir de détection de l'analyse de signature pour les erreurs du code application. L'amélioration du taux de détection au niveau des erreurs injectées dans le système, par contre, est moins sensible que ce qui avait été évalué, ceci d'une part à cause des conditions de mesures et d'autre part à cause de

l'efficacité de la détection des erreurs du système par le 80386, même utilisé en mode réel. Néanmoins, les expériences ont montré que des erreurs du système étaient détectées par les dispositifs implantés, dont une partie d'erreurs perturbatrices, et ceci malgré le peu de vérifications faites pour détecter les erreurs du système. Par ailleurs, certaines erreurs dangereuses de l'UC sont interceptées par l'analyse de signature avant le déclenchement du chien de garde temporel. Les dispositifs implantés vont donc bien dans le sens d'une amélioration de la sûreté du système.

4.6. Conclusion sur l'implantation

L'introduction de dispositifs de test en ligne supplémentaires dans une UC d'automate programmable a conduit à l'implantation d'un mécanisme de vérification du flot de contrôle de l'UC à deux niveaux. Le premier niveau de vérification est réalisé grâce à une signature dérivée sur le code de l'application exécutée par l'UC, et le second niveau correspond à une signature imposée sur certaines parties du système de l'UC. Un dispositif d'inhibition des entrées/sorties a de plus été implanté afin d'assurer qu'aucune erreur détectable par un dispositif de latence supérieure à un cycle de tâche ne soit propagée vers le processus contrôlé.

L'efficacité de ces dispositifs a été démontrée par des expériences d'injections de fautes. Compte tenu du faible coût matériel et logiciel de ces dispositifs, de l'absence de modifications sur l'architecture de l'UC et sur le programme d'application exécuté, les dispositifs implantés présentent un intérêt certain et l'utilisation de ces techniques dans un produit est tout à fait envisageable.

Conclusion générale

La première contribution de cette thèse concerne la description des principales méthodes de vérification du flot de contrôle d'un microprocesseur avec leurs avantages et inconvénients sur différents aspects tels que difficulté de mise en oeuvre et efficacité du test. De cette étude, il ressort que la plupart de ces méthodes nécessitent une modification du programme vérifié, ce qui a comme inconvénient de ralentir le fonctionnement du système et de poser des problèmes de compatibilité logicielle.

La seconde contribution concerne la définition de deux nouvelles méthodes de vérification du flot de contrôle d'un processeur dont la particularité est de ne pas nécessiter la modification du programme vérifié (méthodes DSM : Disjoint Signature Monitoring), remédiant ainsi aux défauts précédemment mentionnés.

La première de ces deux méthodes (WDP) est très facilement adaptable à différents microprocesseurs, et son efficacité est particulièrement bonne, en particulier en ce qui concerne les erreurs de séquencement et la latence de détection. Deux exemples de circuits de processeurs watchdog WDP ont été conçus, le premier pour un Intel 80386sx et le second pour un MC68000. De ces implantations, il ressort que le coût matériel d'un processeur watchdog pour cette méthode est de l'ordre de 4500 portes, ce qui permet de l'intégrer dans un ASIC de petite dimension. Le coût au niveau d'une carte est donc assez réduit. Le watchdog spécifique pour MC68000 a été modélisé en VHDL pour être simulé dans son environnement. Ceci a permis de valider d'une part les différents aspects de l'implantation de cette méthode et d'autre part le fonctionnement du watchdog dans un maximum de situations.

La seconde méthode DSM proposée (DJAM), dérivée de WDP, permet de diminuer sensiblement le coût mémoire des informations nécessaires à la vérification. Cette méthode a été implantée sur un prototype d'unité centrale d'automate programmable, avec un coût matériel particulièrement réduit.

En effet, le prototype d'unité centrale reste quasiment inchangé matériellement, seul un composant spécifique ayant été modifié avec un surcoût matériel de l'ordre de 11%. Au niveau du système ceci est très faible, d'autant plus que l'architecture générale de l'UC n'est pas affectée et qu'il n'est pas nécessaire de modifier la carte du système. Par ailleurs, l'absence de modifications sur le programme d'application de l'automate rend l'utilisation de ces mécanismes de test en ligne totalement transparente vis à vis de l'utilisateur.

Des expériences d'injections de fautes effectuées sur ce prototype ont montré l'apport de cette méthode en matière de détection des erreurs affectant le déroulement du programme d'application de l'automate et le fonctionnement du système de l'unité centrale. Ces expériences d'injection de fautes apportent également des éléments sur l'efficacité relative de plusieurs dispositifs de test en ligne, présents dans de nombreux systèmes.

Compte tenu des caractéristiques de ce prototype d'UC, les dispositifs implantés présentent un intérêt certain et l'adaptation de ces techniques pour un produit industriel peut être envisagée.

Références bibliographiques

[Abou 90] P. Abouzeid, R. Leveugle, T. Michel, K. Sakouti, G. Saucier,
"Partitioning and logic synthesis",
IFIP workshop on partitioning, Munich, F.R.G., January 29-30, 1990.

[Ayac 79] J. M. Ayache, P. Azéma, M. Diaz,
"Observer : a concept for detection of control errors in concurent systems",
9th Fault Tolerant Computing Symposium (FTCS), 1979, pp. 79-85.

[Arla 89] J. Arlat, Y. Crouzet, J. C. Laprie,
"Fault injection for dependability validation of fault-tolerant computing systems",
19th Fault Tolerant Computing Symposium (FTCS), 1989, pp. 348-355.

[Arla 90a] J. Arlat,
"Validation de la sûreté de fonctionnement par injection de fautes : méthode - mise en oeuvre - application",
Thèse d'état, Toulouse, France, Décembre 1990.

[Arla 90b] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell,
"Fault injection for dependability validation: a methodology and some applications",
IEEE trans. on Software Eng., vol. 16, n° 2, February 1990, pp. 166-182.

[Chil 89] R. Chillarege, N.S. Bowen,
"Understanding large system failures - A fault injection experiment",
19th Fault Tolerant Computing Symposium (FTCS), 1989, pp 356-363.

[Conn 72] J. R. Connet, E. J. Pasternak, B. D. Wagner,
"Software defenses in real-time control systems",
2nd Fault-Tolerant Computing Symposium (FTCS), 1972, pp. 94-99.

[Czec 90] E.W. Czeck, D.P. Siewiorek,
"Effects of transient gate-level faults on program behavior",
20th Fault Tolerant Computing Symposium (FTCS), 1990, pp. 236-243.

[Dami 88] M. Damiani, P. Olivo, M. Favalli, B. Ricco,
"Aliasing errors in signature analysis testing of integrated circuits",
Int. Conference on Computer Design (ICCD),1988,pp. 458-461.

[Dami 89] M. Damiani et al,
"Aliasing in signature analysis testing with multiple-input shift-registers",
First European Test Conference, 1989, pp. 346-353.

[Damm 86] A. Damm,
"The effectiveness of software error-detection mechanisms in real-time operating systems",
16th Fault Tolerant Computing Symposium (FTCS), 1986, pp 171-176.

[Davi 80] R. David,
"Testing by Feedback Shift Register",
IEEE trans. on Computers, vol. C-29, no. 7, July 1980, pp. 668-673.

[Davi 84] R. David,
"Signature analysis of multi-output circuits",
14th Fault-Tolerant Computing Symposium (FTCS), 1984, pp. 366-371.

[Davi 86] R. David,
"Signature analysis for multiple-output circuits",
IEEE trans. on Computers, vol. C-35, no. 9, September 1986, pp. 830-837.

[Delo 90] X. Delord, G. Saucier,
"Control flow checking in pipelined RISC microprocessors: the Motorola MC88100 case study",
EUROMICRO'90 Workshop on real time, 1990, pp 162-169.

[Delo 91] X. Delord, G. Saucier,
"Formalizing signature analysis for control flow checking of pipelined RISC microprocessors",
International Test Conference (ITC),1991, pp 936-945.

[Delo 93] X. Delord,
"Analyse de signature adaptée aux processeurs RISC",
Thèse en microélectronique INPG, Grenoble, Février 1993.

[Duba 88] P. Duba, R.K. Iyer,
"Transient fault behavior in a microprocessor",
IEEE proc. Int. Conf. on Computer Design (ICCD), 1988, pp 272-276.

[Edmo 90] P. Edmond, A. Gupta, D.P. Siewiorek, A.A. Brennan,
"ASSURE: automated design for dependability",
27th Design Automation Conference, 1990, pp. 555-560.

[Eife 84] J. B. Eifert, J. P. Shen,
"Processor monitoring using asynchronous signatured instruction streams",
14th Fault-Tolerant Computing Symposium (FTCS), 1984, pp.394-399.

[Fent 84] B. P. Fenton, I. M. Grant, D. R. Thiessen,
"Multiple-line signature analysis using parallel linear feedback shift register",
IBM Tech. Disclosure Bulletin, vol. 26, no. 12, May 1984, pp. 6358-6360.

[Gels 87] P. P. Gelsinger,
"Design and test of the 80386",
IEEE Design & Test of Computers, vol. 4, no. 3, June 1987, pp.42-50.

[Gera 86] J. P. Gerardin,
"Aide à la conception d'appareils fiables et sûrs : l'appareil DEFI",
Electronique Industrielle, no. 116, 15 Novembre 1986.

[Gond 85] M. Gondran, M.Minoux,
"Graphes et algorithmes",
Eyrolles, Paris, France, 1985.

[Gunn 89] U.Gunneflo, J.Karlson, J.Torin,
"Evaluation of error detection schemes using fault injection by heavy-ion radiation",
19th Fault Tolerant Computing Symposium (FTCS), 1989, pp 340-347.

[Harw 89] W. Harwood, M. McDermott,
"Testability features of the MC68332 modular microcontroller",
International Test Conference (ITC), 1989, pp. 615-623.

[Hass 82] S. Z. Hassan,
"Algebraic analysis of parallel signature analyzers",
Center for Reliable Computing Technical Report No. 82-5, Stanford University, California, USA, June 1982.

[Hass 84] S. Z. Hassan, E. J. McCluskey,
"Enhancing the effectiveness of parallel signature analyzers",
International Conference on Computer-Aided Design (ICCAD), 1984, pp. 102-104.

[Holm 91] L. Holmquist, L.L. Kinney,
"Concurent error detection for restricted fault sets in sequential circuits and microprogrammed control units using convolutional codes",
International Test Conference (ITC), 1991, pp 926-935.

[Hort 90] P. D. Hortensius, R. D. McLeod, H. C. Card,
"Cellular automata-based signature analysis for built-in self-test",
IEEE trans on Computers, vol. 39, no. 10, October 1990, pp. 1273-1283.

[Ivan 87] A. Ivanov, V. Agarwal,
"On a fast method to monitor the behaviour of signature analysis registers",
International Test Conference (ITC), 1987, pp. 645-655.

[Ivan 88] A. Ivanov, V. K. Agarwal,
"An iterative technique for calculating aliasing probability of linear feedback signature registers",
18th Fault-Tolerant Computing Symposium (FTCS), 1988, pp. 70-75.

[Ivan 92] A. Ivanov, S. Pilarski,
"Performance of signature analysis: a survey of bounds, exact, and heuristic algorithms",
Integration, the VLSI journal, vol 13, n°1, May 92, pp 17-38.

[Iyen 85] V. S. Iyengar, L. L. Kinney,
"Concurrent fault detection in microprogrammed control units",
IEEE trans. on Computers, vol. C-34, n° 9, September 1985, pp. 810-821.

[Jay 86] C. Jay,
"HSURF, un microprocesseur facilement testable pour des applications à haute sûreté de fonctionnement",
Thèse en microélectronique INPG, Grenoble,Juin 1986.

[Kana 92] G.A. Kanawati, N.A. Kanawati, J.A. Abraham,
"FERRARI: A tool for the validation of dependability properties",
22th Fault Tolerant Computing Symposium (FTCS), 1992, pp 336-344.

[Karl 91] J. Karlson, U. Gunneflo, P. Lidén, J. Torin,
"Two fault injection techniques for test of fault handling mechanisms",
International Test Conference (ITC), 1991, pp 140-149.

[Lapr 85] J.C.Laprie,
"Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes",
Tech. et Sciences Informatiques, 4 (5), pp. 419-429, Sept.-Oct. 1985.

[Lapr 92] J. -C. Laprie (Ed.),
"Dependability: Basic Concepts and Terminology",
Dependable Computing and Fault Tolerance, 5, 265p.,
ISBN N°3-211-82296, Springer-Verlag, Vienna, 1992

[Leve 90a] R. Leveugle,
"Analyse de signature et test en ligne intégré sur silicium",
Thèse en microélectronique INPG, Grenoble, Janvier 1990.

[Leve 90b] R. Leveugle, G. Saucier,
"Optimized synthesis of concurrently checked controllers",
IEEE transactions on Computers, vol. 39, no. 4, April 1990, pp. 419-425.

[Leve 90c] R. Leveugle, T. Michel, G. Saucier,
"Design of microprocessors with built-in on-line test",
20th Fault-Tolerant Computing Symposium (FTCS), 1990, pp. 450-456.

[Leve 90d] R. Leveugle, T. Michel, G. Saucier,
"Design of an application specific microprocessor with built-in on-line test capabilities",
IFIP Workshop on Design and Test of ASICs, Hiroshima, Japan, 1990, pp. 7-10.

[Lu 82] D.J.Lu,
"Watchdog processor and structural integrity checking",
IEEE trans on Comp., vol C-31,n° 7, July 1982, pp 681-685.

[Made 91a] H. Madeira, J.G. Silva,
"On-line signature learning and checking",
2nd Int. Conf. on Dependable Computing for Critical Application, 1991, pp 170-177.

[Made 91b] H. Madeira, J.G. Silva,
"On-line signature learning and checking: experimental evaluation",
IEEE proc. COMPEURO, 1991, pp 642-646.

[Mahm 83] A. Mahmood, E. J. McCluskey, D. J. Lu,
"Concurrent fault detection using a watchdog processor and assertions",
International Test Conference (ITC), 1983, pp. 622-628.

[Mahm 85a] A. Mahmood, A. Ersoz, E. J. McCluskey,
"Concurrent system-level error detection using a watchdog processor",
International Test Conference (ITC), 1985, pp. 145-152.

[Mahm 85b] A. Mahmood, E. J. McCluskey,
"Watchdog processors: error coverage and overhead",
15th Fault-Tolerant Computing Symposium (FTCS), 1985, pp. 214-219.

[Mahm 88] A. Mahmood, E. J. McCluskey,
"Concurrent error detection using watchdog processors - A survey",
IEEE trans. on Comp., vol. 37, n°.2, February 1988, pp. 160-174.

[Mart 90] J. Martin, S. Wartski, C. Galivel,
"Le Processeur Codé : un nouveau concept appliqué à la sécurité des systèmes de transports",
Revue Générale des Chemins de Fer, juin 1990, pp 29-35.

[McAn 87] W. H. McAnney, J. Savir,
"There is information in faulty signatures",
International Test Conference (ITC), 1987, pp. 630-636.

[Mich 90] G. Michel,
"Programmable logic controllers: architecture and applications",
Ed. John Wiley & Sons, pp. 93-108.

[Mich 91] T. Michel, R. Leveugle, G. Saucier,
"A new approach to control flow checking without program verification",
21st Fault-Tolerant Computing Symposium (FTCS), 1991, pp 334-341.

[Mich 92a] T. Michel, R. Leveugle, F. Gaume, R. Roane,
"An Application Specific Microprocessor with Two-Level Built-In Control Flow Checking Capabilities",
EURO-ASIC'92, Paris, France, June 1-5, 1992, pp 310-313.

[Mich 92b] T. Michel, R. Leveugle, X. Delord, G. Saucier,
"Analyse de signature: contraintes liées à l'architecture des processeurs",
8ième Colloque de fiabilité et de maintenabilité, Grenoble, France, 6-8 octobre 1992, pp 636-643.

[Mire 92] G. Miremadi, J. Karlson,
"Two software techniques for on-line error detction",
22th Fault Tolerant Computing Symposium (FTCS), 1992, pp 328-335.

[Namj 82a] M. Namjoo,
"Techniques for concurrent testing of VLSI processor operation",
International Test Conference (ITC), 1982, pp.461-468.

[Namj 82b] M. Namjoo, E.J.McCluskey,
"Watchdog processors and capability checking",
12th Fault-Tolerant Computing Symposium (FTCS), 1982, pp 245-248.

[Namj 82c] M. Namjoo,
"Design of concurrently testable microprogrammed control units",
15th Microprogramming Workshop (Micro 15), 1982, pp.173-180.

[Namj 83] M. Namjoo,
"CERBERUS-16 : an architecture for a general purpose watchdog processor",
13rd Fault-Tolerant Computing Symposium (FTCS), 1983, pp. 216-219.

[Ohls 92] J. Ohlsson, M. Rimén, U. Gunneflo,
"A study of the effect of transient fault injection into a 32-bit RISC with built-in watchdog",
22th Fault Tolerant Computing Symposium (FTCS), 1992, pp 316-325.

[Oliv 89] P. Olivo, M. Damiani, B. Ricco,
"On the design of multiple-input shift-registers for signature analysis testing",
International Test Conference (ITC), 1989, pp. 936.

[Robi 92a] S. H. Robinson, J. P. Shen,
"Direct methods for synthesis of self-monitoring state machines",
22th Fault Tolerant Computing Symposium (FTCS), 1992, pp. 306-315.

[Robi 92b] S. H. Robinson,
"Finite-state machine synthesis for continuous, concurrent error detection using signature-invariant monitoring"
PhD dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 1992.

[Saxe 89] N. R. Saxena, E. J. McCluskey,
"Control-flow checking using watchdog assists and extended-precision checksums",
19th Fault-Tolerant Computing Symposium (FTCS), 1989, pp. 428-435.

[Saxe 90] N. R. Saxena, E. J. McCluskey,
"Control-flow checking using watchdog assists and extended-precision checksums",
IEEE transactions on Computers, vol. 39, no. 4, April 1990, pp. 554-559.

[Schm 82] M.E.Schmid,R.L.Trapp,A.E.Davidoff,G.M.Masson,
"Upset exposure by means of abstractions",
12th Fault Tolerant Computing Symposium (FTCS), 1982, pp 237-244.

[Schu 86] M. A. Schuette, J. P. Shen, D. P. Siewiorek, Y. X. Zhu,
"Experimental evaluation of two concurrent error detection schemes",
16th Fault Tolerant Computing Symposium (FTCS), 1986, pp. 138-143.

[Schu 87] M. A. Schuette, J. P. Shen,
"Processor control flow monitoring using signatured instruction streams",
IEEE trans. on Computers, vol. 36, n° 3, March 1987, pp. 264-276.

[Schu 91] M.A. Shuette, J.P. Shen,
"Exploiting instruction-level resource parallelism for transparent integrated control-flow monitoring",
21st Fault-Tolerant Computing Symposium (FTCS), 1991, pp 318-325.

[Schw 87] A.Schweitzer,
"Amélioration du niveau de sécurité des systèmes électroniques programmables par application du concept d'analyse de signature",
Thèse en Automatique Nancy I, Mai 1987.

[Shen 83] J. P. Shen, M. A. Schuette,
"On-line self-monitoring using signatured instruction streams",
International Test Conference (ITC), 1983, pp. 275-282.

[Smit 80] J. E. Smith,
"Measures of effectiveness of fault signature analysis",
IEEE trans. on Computers, vol. C-29, n°6, June 1980, pp. 510-514.

[Sosn 88] J. Sosnowski,
"Detection of control flow errors using signature and checking instructions",
International Test Conference (ITC), 1988, pp. 81-88.

[Srid 82] T. Sridhar, S. M. Thatte,
"Concurrent checking of program flow in VLSI processors",
International Test Conference (ITC), 1982, pp. 191-199.

[Toma 85] S.P. Tomas, J.P. Shen,
"A roving monotoring processor for detection of control flow errors in multiple processor systems",
IEEE proc. Int. Conf. Computer Design (ICCD), 1985, pp 531-539.

[Tung 86] C.-H. Tung, J. P. Robinson,
"On concurrent testable microprogrammed control units",
International Test Conference (ITC), 1986, pp. 895-900.

[Upad 91] J. S. Upadaya, B. Ramamurthy,
"A new efficient signature technique for process monitoring in critical systems",
2nd Int. Conf. on Dependable Computing for Critical Application, 1991, pp 178-179.

[Wart 90] N. J. Warter, W. W. Hwu,
"A software based approach to achieving optimal performance for signature control flow checking",
20th Fault-Tolerant Computing Symposium (FTCS), 1990, pp. 442-449.

[Wilk 87] K. D. Wilken, J. P. Shen,
"Embedded Signature Monitoring: analysis and technique",
International Test Conference (ITC), 1987, pp. 324-333.

[Wilk 88] K. Wilken, J. P. Shen,
"Continuous signature monitoring : efficient concurrent-detection of processor control errors",
International Test Conference (ITC), 1988, pp. 914-925.

[Wilk 90] K. Wilken, J. P. Shen,
"Continuous signature monitoring: low-cost concurrent detection of processor control errors",
IEEE transactions on Computer-Aided Design, vol. 9, n° 6, June 1990, pp 629-641.

[Wilk 91] K. Wilken,
"Optimal Signature Placement for Processor-Error Detection using Signature Monitoring",
21st Fault Tolerant Computing Symposium (FTCS), 1991, pp 326-333.

[Will 86] T. W. Williams, W. Daehn, M. Gruetzner, C. W. Starke,
"Comparison of aliasing errors for primitive and non-primitive polynomials",
International Test Conference (ITC), 1986, pp. 282-288.

[Will 87a] T. W. Williams, W. Daehn, M. Gruetzner, C. W. Starke,
"Aliasing errors with primitive and non-primitive polynomials",
International Test Conference (ITC), 1987, pp. 637-644.

[Will 87b] T. W. Williams, W. Daehn, M. Gruetzner, C. W. Starke,
"Aliasing errors in signature analysis registers",
IEEE Design & Test of Computers, vol. 4, no. 2, April 1987, pp. 39-45.

[Will 88] T. W. Williams, W. Daehn, M. Gruetzner, C. W. Starke,
"Bounds and analysis of aliasing errors in linear feedback shift registers",
IEEE transactions on Computer-Aided Design, vol. 7, no. 1, January 1988, pp. 75-83.

[Will 89] T. W. Williams, W. Daehn,
"Aliasing errors in multiple input signature analysis registers",
First European Test Conference, April 1989, pp. 338-345.

[Xavi 89] D. Xavier, R. C. Aitken, A. Ivanov, V. K. Agarwal,
"Experiments on aliasing in signature analysis registers",
International Test Conference (ITC), 1989, pp. 344-354.

[Yau 80] S. S. Yau, F.C. Chen,
"An approach to concurrent control flow checking",
IEEE transactions on Software Engineering, vol. SE-6, no. 2, March 1980, pp. 126-137.

ANNEXES

1. Fonctionnement du watchdog WDP pour chaque type de noeud

Par soucis de simplification, la description de la méthode WDP faite dans cette section ne tient pas compte de la présence éventuelle d'une lecture anticipée des instructions dans le processeur vérifié. De même, les problèmes liés aux exceptions sont ignorés.

Dans cette section, les notations suivantes seront utilisées :

- Aμ : adresse courante du microprocesseur,
- Aμi : adresse dans le programme microprocesseur du noeud i,
- A_{wi} : adresse dans le programme watchdog du noeud i.

Le watchdog dont le fonctionnement est ici décrit nécessite l'utilisation des registres suivants :

- PCW : pointeur d'instruction watchdog,
- SIGN : registre de signature courante,
- SPW : pointeur de la pile du watchdog,
- MEMO : registre intermédiaire de mémorisation du PCW.

1.1. Initialisation du processeur watchdog

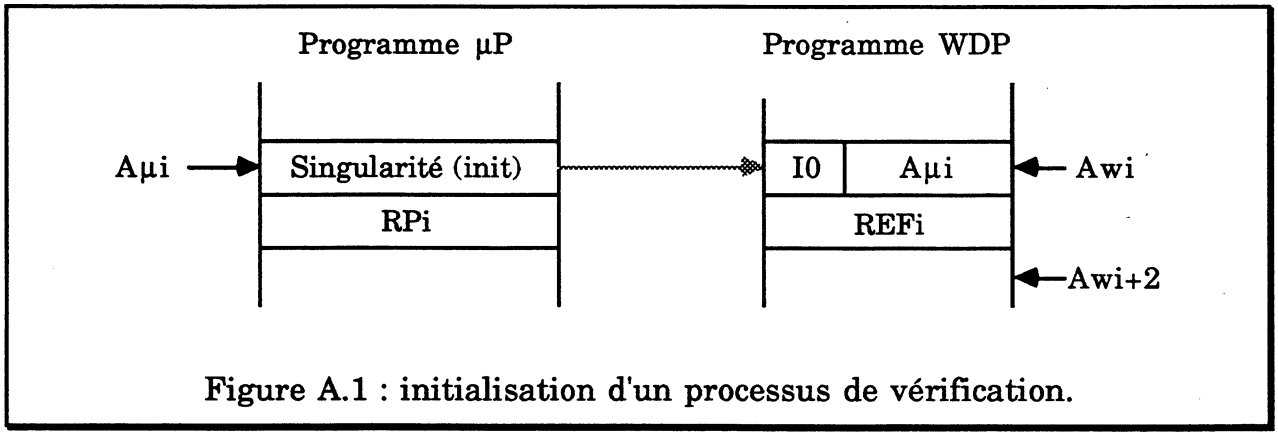


Figure A.1 : initialisation d'un processus de vérification.

Si Iμ ≠ instruction(init) alors ALARME
PCW ← RPi
si Aμ ≠ Aμi + 1 alors ALARME
sinon SIGN ← REFi
attendre Aμ = Aμ(PCW)

1.2. Début de bloc linéaire

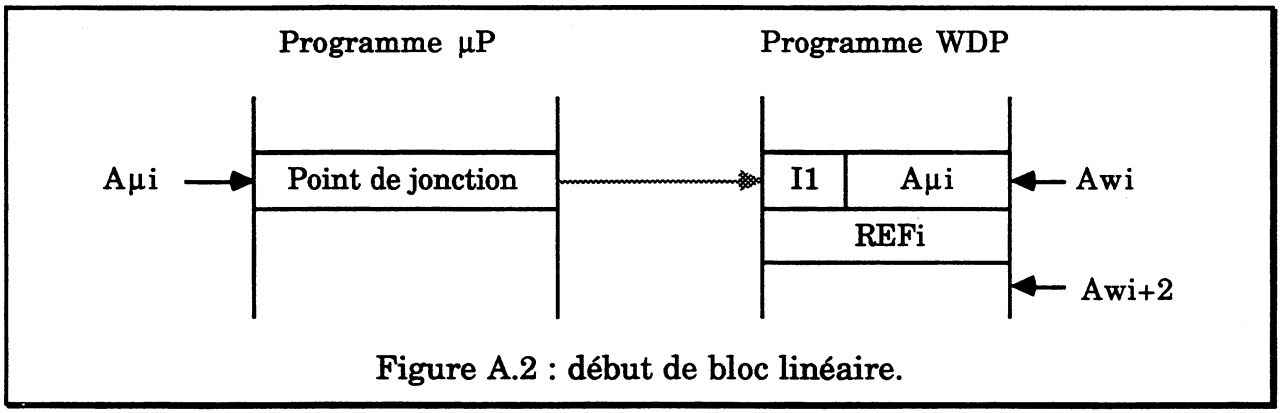


Figure A.2 : début de bloc linéaire.

quand Aµ = Aµi faire
si SIGN ≠ REFi alors ALARME
sinon PCW ← PCW + 2
attendre Aµ = Aµ(PCW)

1.3. Branchement inconditionnel

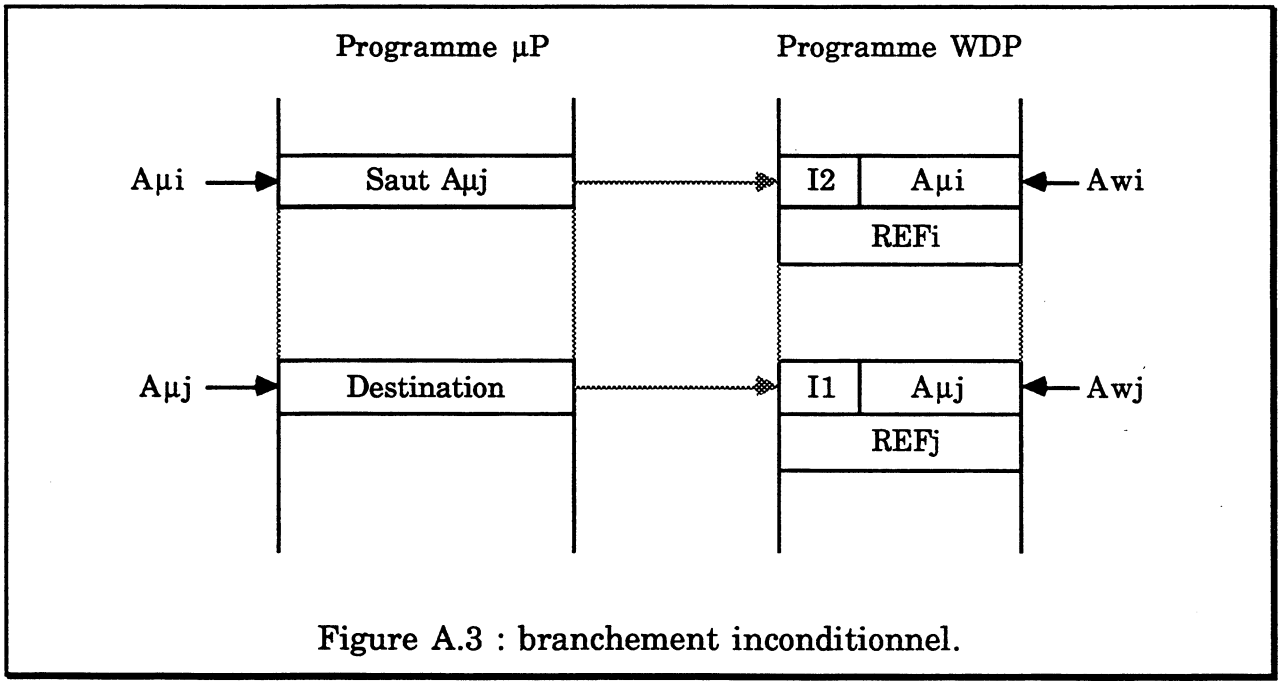


Figure A.3 : branchement inconditionnel.

quand Aµ = Aµi faire
PCW ← SIGN ⊕ AJSi (PCW = Awj)
attendre saut µP
si Aµ ≠ Aµj alors ALARME
sinon SIGN ← REFj
PCW ← PCW + 2
attendre Aµ = Aµ(PCW)

1.4. Branchement conditionnel

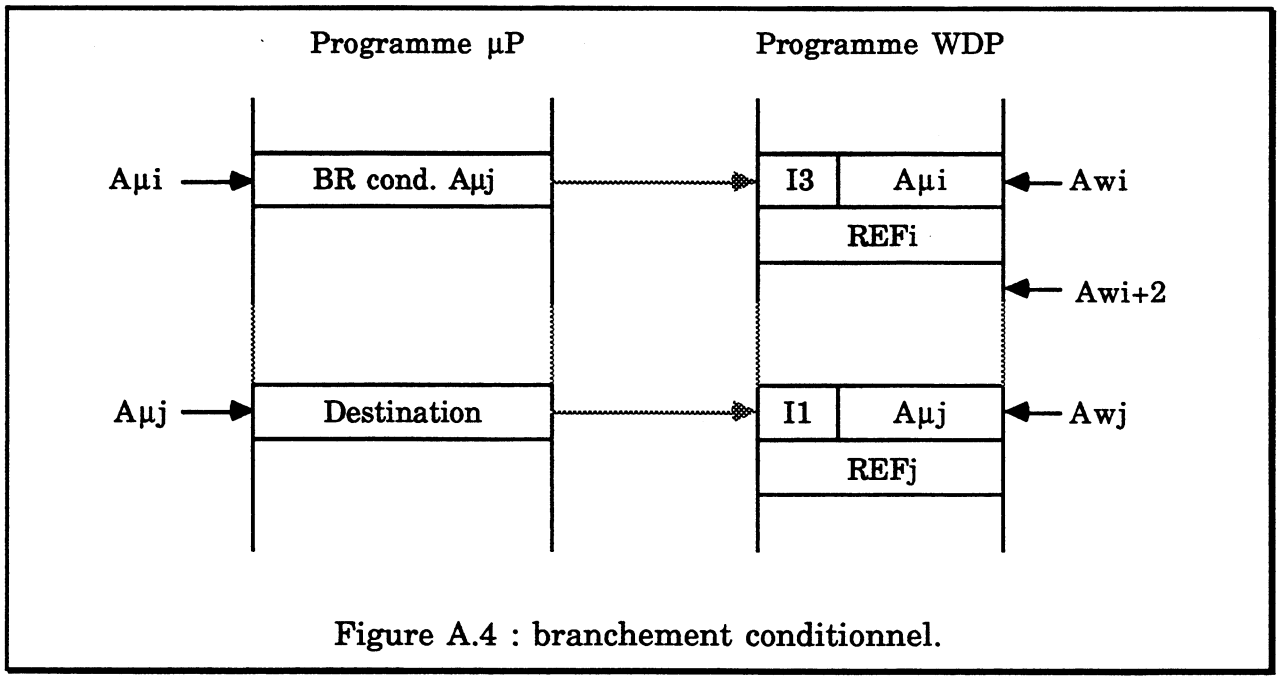


Figure A.4 : branchement conditionnel.

quand Aμ = Aμi faire
MEMO ← PCW (MEMO = Awi)
PCW ← SIGN ⊕ REFi (PCW = Awj)
si saut μP alors
si Aμ ≠ Aμj alors ALARME
sinon SIGN ← REFj
PCW ← PCW + 2
sinon PCW ← MEMO + 2
attendre Aμ = Aμ(PCW)

1.5. Appel et retour inconditionnels de sous-programme

a/ Appel inconditionnel de sous-programme

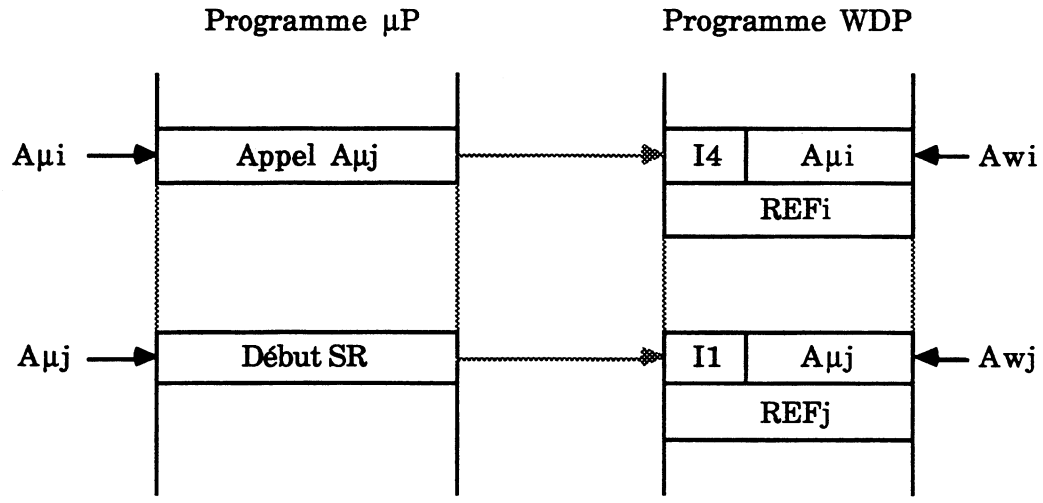


Figure A.5 : appel inconditionnel de sous-programme.

quand Aμ = Aμi faire
PILE(SPW++) ← PCW
PCW ← SIGN ⊕ REFi (PCW = Awj)
attendre saut μP
si Aμ ≠ Aμj alors ALARME
sinon SIGN ← REFj
PCW ← PCW + 2
attendre Aμ = Aμ(PCW)

b/ Retour inconditionnel de sous-programme

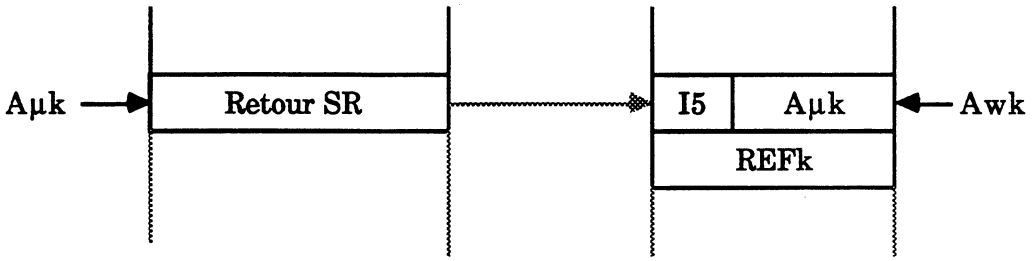


Figure A.6 : retour inconditionnel de sous-programme.

quand Aμ = Aμk faire
si SIGN ≠ REFk alors ALARME
SPW ← SPW -1
PILE(SPW) ← PCW (PCW = Awi : appel sous-programme)
attendre saut μP
si Aμ ≠ Aμi + 1 alors ALARME
sinon SIGN ← REFi
PCW ← PCW + 2
attendre Aμ = Aμ(PCW)

1.6. Appel et retour conditionnels de sous-programme

a/ Appel conditionnel de sous-programme

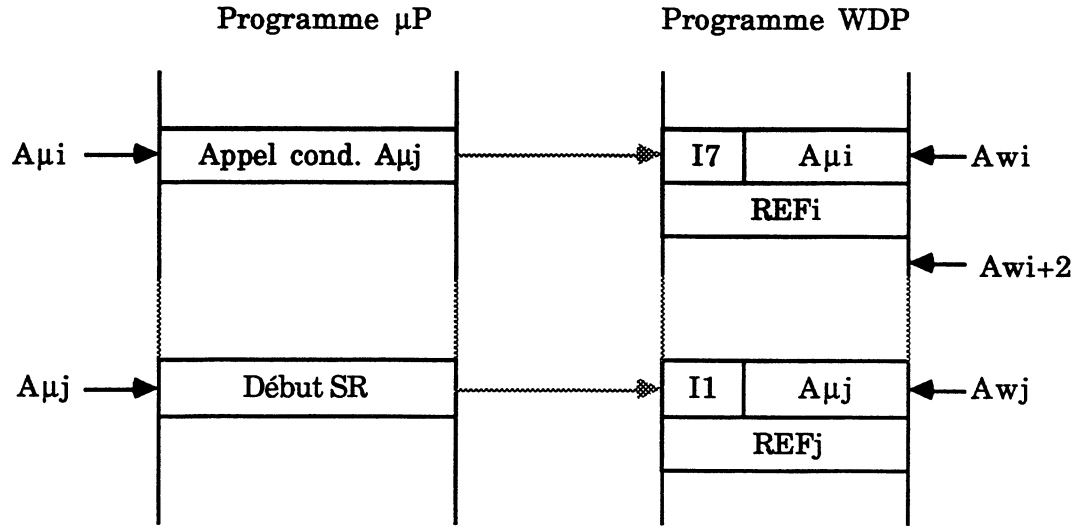


Figure A.7 : appel conditionnel en sous-programme.

```
quand Aµ = Aµi faire
        MEMO ← PCW                          (MEMO = Awi)
        PCW ← SIGN ⊕ REFi                   (PCW = Awj)
        si saut µP alors
                si Aµ ≠ Aµj alors ALARME
                sinon   PILE(SPW++) ← MEMO + 2
                        PILE(SPW++) ← SIGN
                        SIGN ← REFj
                        PCW ← PCW + 2
        sinon   PCW ← MEMO + 2
attendre Aµ = Aµ(PCW)
```

b/ Retour conditionnel de sous-programme

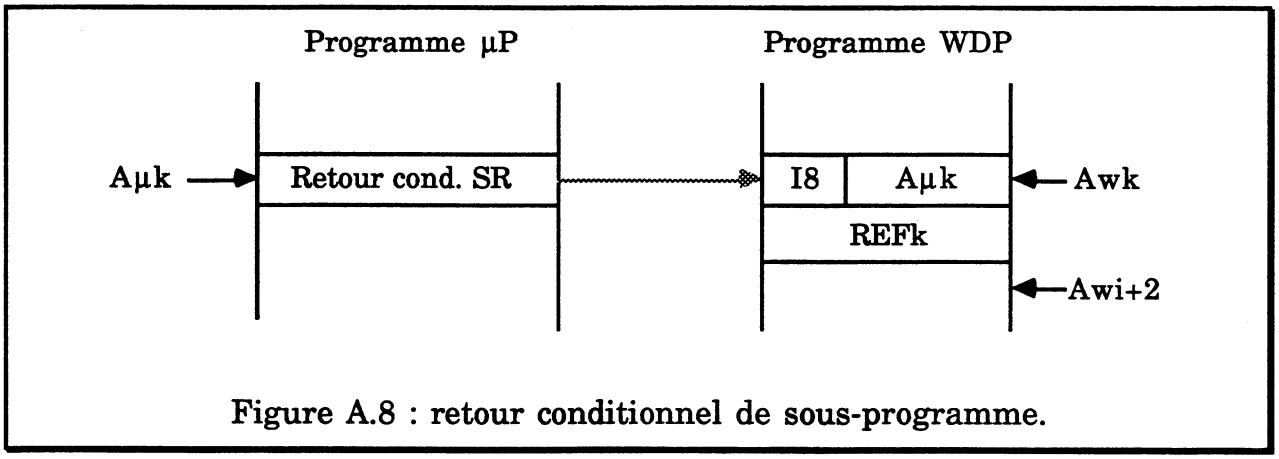
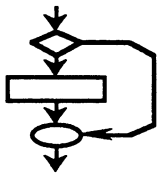


Figure A.8 : retour conditionnel de sous-programme.

```
quand Aμ = Aμk faire
        si SIGN ≠ REFk alors ALARME
        MEMO ← PCW
        SPW ← SPW - 1
        PCW ← PILE(SPW)          (PCW = Awi : appel sous-programme)
        si saut μP alors
                si Aμ ≠ Aμi + 1 alors ALARME
                sinon   SPW ← SPW - 1
                        SIGN ← PILE(SPW)
                        PCW ← PCW + 2
        sinon   SPW ← SPW + 1
                PCW ← MEMO + 2
attendre Aμ = Aμ(PCW)
```

2. WDP et les structures HLL classiques

2.1. Structure "If Then"



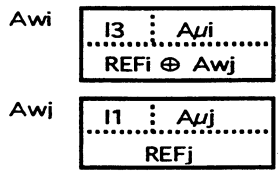
Schéma

Aμi ..... Br If --> Aμj

Then

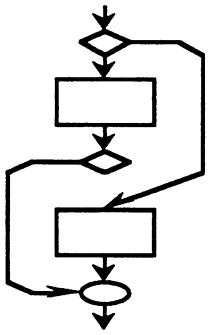
Aμj ..... Inst j

Programme μProc.



Prog. WatchDog

2.2. Structure "If Else"



Schéma

Aμi ..... Br if --> Aμj

Then

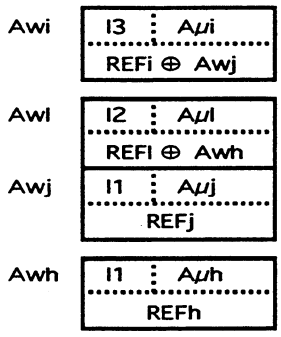
Aμl ..... Saut --> Aμh

Aμj ..... Inst j

Else

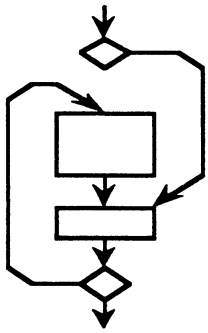
Aμh ..... Inst h

Programme μProc.



Prog. WatchDog

2.3. Structure "While For"



Schéma

Aμi ..... Saut --> Aμj

Aμl ..... Inst l

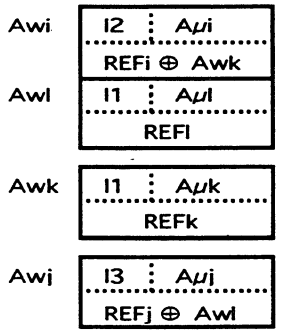
Boucle

Aμk ..... Inst k

Cond.

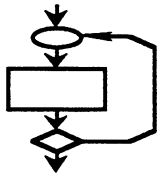
Aμj ..... Br if --> Aμl

Programme μProc.



Prog. WatchDog

2.4. Structure "Repeat Until"



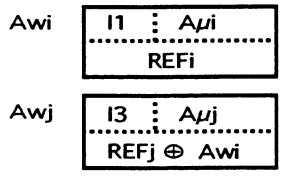
Schéma

Aμi ..... Inst i

Boucle

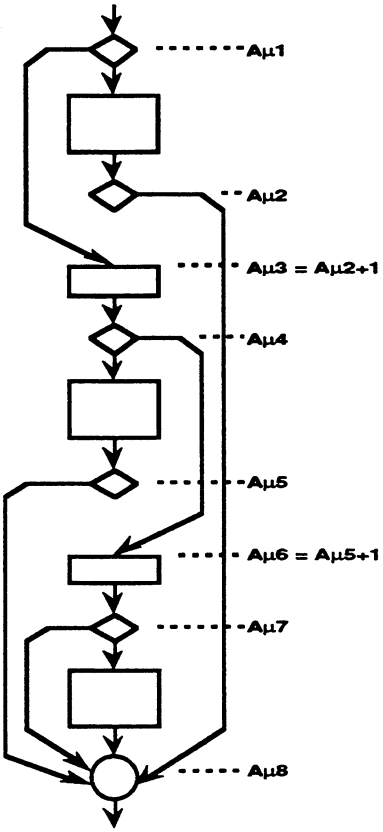
Aμj ..... Br if --> Aμi

Programme μProc.



Prog. WatchDog

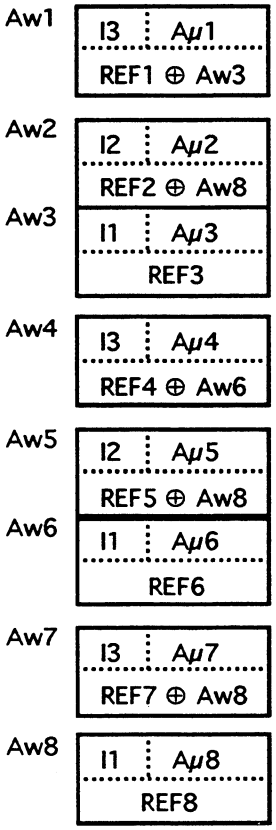
2.5. Structure "Switch"



Schéma

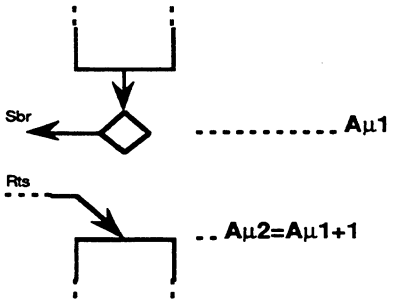
```
Aμ1 ..... Br if --> Aμ3     Cond 1
                            Cas n°1
Aμ2 ..... Saut --> Aμ8
Aμ3 ..... Instr
                            Cond 2
Aμ4 ..... Br if --> Aμ6
                            Cas n°2
Aμ5 ..... Saut --> Aμ8
Aμ6 ..... Instr
                            Cond 3
Aμ7 ..... Br if --> Aμ8
                            Cas n°3
Aμ8 ..... Instr
```

Programme μProc.



Prog. WatchDog

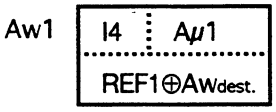
2.6. Structure "Call"



Schéma

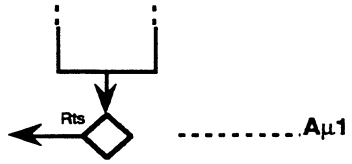
```
Aμ1 ..... Sbr --> Dest.
Aμ2 ..... Instr.
```

Programme μProc.



Prog. WatchDog

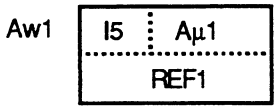
2.7. Structure "Return"



Schéma

```
                            Sous-Progr.
Aμ1 ..... Rts
```

Programme μProc.



Prog. WatchDog

3. Processeur WDP pour MC68000

3.1. Caractéristiques générales du MC68000

Le MC68000 est un microprocesseur 16/32 bits, c'est à dire qu'il possède une architecture interne sur 32 bits mais un bus de donnée sur 16 bits seulement.

La fréquence de fonctionnement du MC68000 peut varier de 4 à 16 MHz selon la version et la technologie.

Le bus adresses est sur 23 bits (A1 à A23) et adresse des mots de 16 bits.

L'octet est adressable par l'intermédiaire de deux signaux (UDS et LDS), qui sont dérivés du 1er bit de l'adresse (A0).

Les principaux signaux d'un MC68000 sont représentés en Figure A.9 .

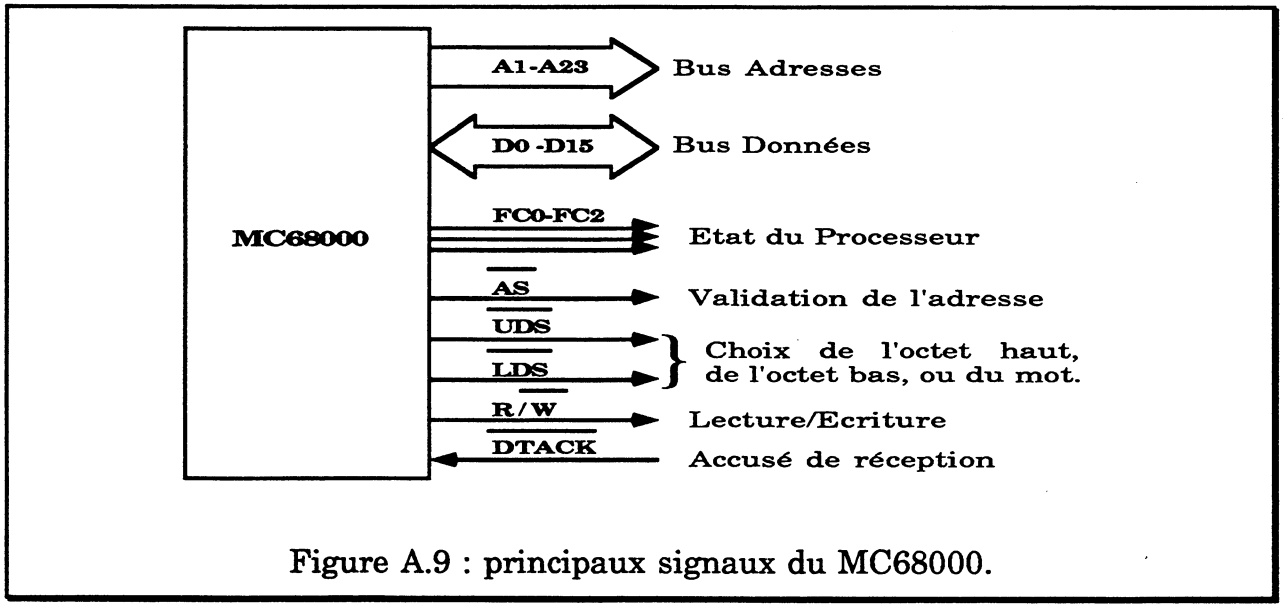


Figure A.9 : principaux signaux du MC68000.

Trois signaux, **FC0 - FC2**, indiquent l'état du processeur. FC0 au niveau bas et FC1 au niveau haut indiquent que le microprocesseur est en train de lire du code programme. FC0 - FC2 au niveau haut indiquent que le microprocesseur est en cours de reconnaissance d'une exception.

Les échanges avec la mémoire centrale se font de manière asynchrone.

Le signal **R/W** (Read/Write) indique le sens de l'échange (lecture ou écriture).

Les signaux **UDS** et **LDS** (Upper, Lower Data Strobe) indiquent si on adresse le mot, l'octet haut ou l'octet bas.

Le signal **AS** (Address Strobe) indique que l'adresse présente sur le bus d'adresses est valide, et est actif au niveau bas.

Le signal **DTACK** (Data Transfert ACKnowledge), actif au niveau bas, indique au microprocesseur s'il peut terminer l'échange mémoire en cours. Tant que ce signal n'est pas actif (bas) pendant un cycle d'accès mémoire, le MC68000 insère des cycles d'attente pour allonger la durer de l'accès mémoire en cours.

Le MC68000 possède beaucoup d'autres caractéristiques et signaux, mais ce qui a été présenté ici suffit à la compréhension des paragraphes qui suivent. Ce qui ressort de la description donnée ci-dessus est qu'il est facile de savoir quand le microprocesseur charge une instruction, quand il part en exception ou encore qu'il est possible de le bloquer en agissant sur le signal DTACK. Ainsi, les signaux décrits ici vont pouvoir être utilisés par le processeur watchdog pour se synchroniser avec le microprocesseur.

Le MC68000 possède un format d'instruction relativement simple. La taille des instructions peut varier de 1 à 5 mots mémoire, le code opération étant toujours un mot (ni plus, ni moins). Les mots supplémentaires représentent des valeurs immédiates ou des adresses d'opérandes.

3.2. Le Processeur Watchdog

Le processeur Watchdog conçu pour le MC68000 possède la même architecture que celle décrite en section 2.7.1 . On retrouve les 4 mêmes modules. La mémoire utilisée par le watchdog est une mémoire organisée en mots de 16 bits et séparée de la mémoire centrale du MC68000 (mémoire locale au processeur watchdog).

3.2.1. Schéma général et signaux

Le schéma en Figure A.10 précise l'architecture générale du processeur watchdog pour un MC68000, avec les signaux échangés entre les différents modules. Un rapide commentaire des principaux signaux apparaissant sur cette figure est donné ci-dessous.

Bus Adresses et Bus Données : bus d'adresses du microprocesseur sur 23 bits et bus de données sur 16 bits. Le bus de données est connecté à l'interface microprocesseur car c'est elle qui est chargée de faire l'interface entre le module Exécuteur et le 68000 en ce qui concerne les exceptions.

Exec I, Exec C, Exec @ et Exec D : bus sur 16 bits chargés d'acheminer les données entre les différents modules. Exec I et Exec @ peuvent être connectés entre eux, ainsi que Exec C et Exec D et traversent alors le module exécuteur.

Adresses WDP et Données WDP : bus sur 16 bits pour les échanges avec la mémoire. Seule la partie Interface Mémoire en a l'accès.

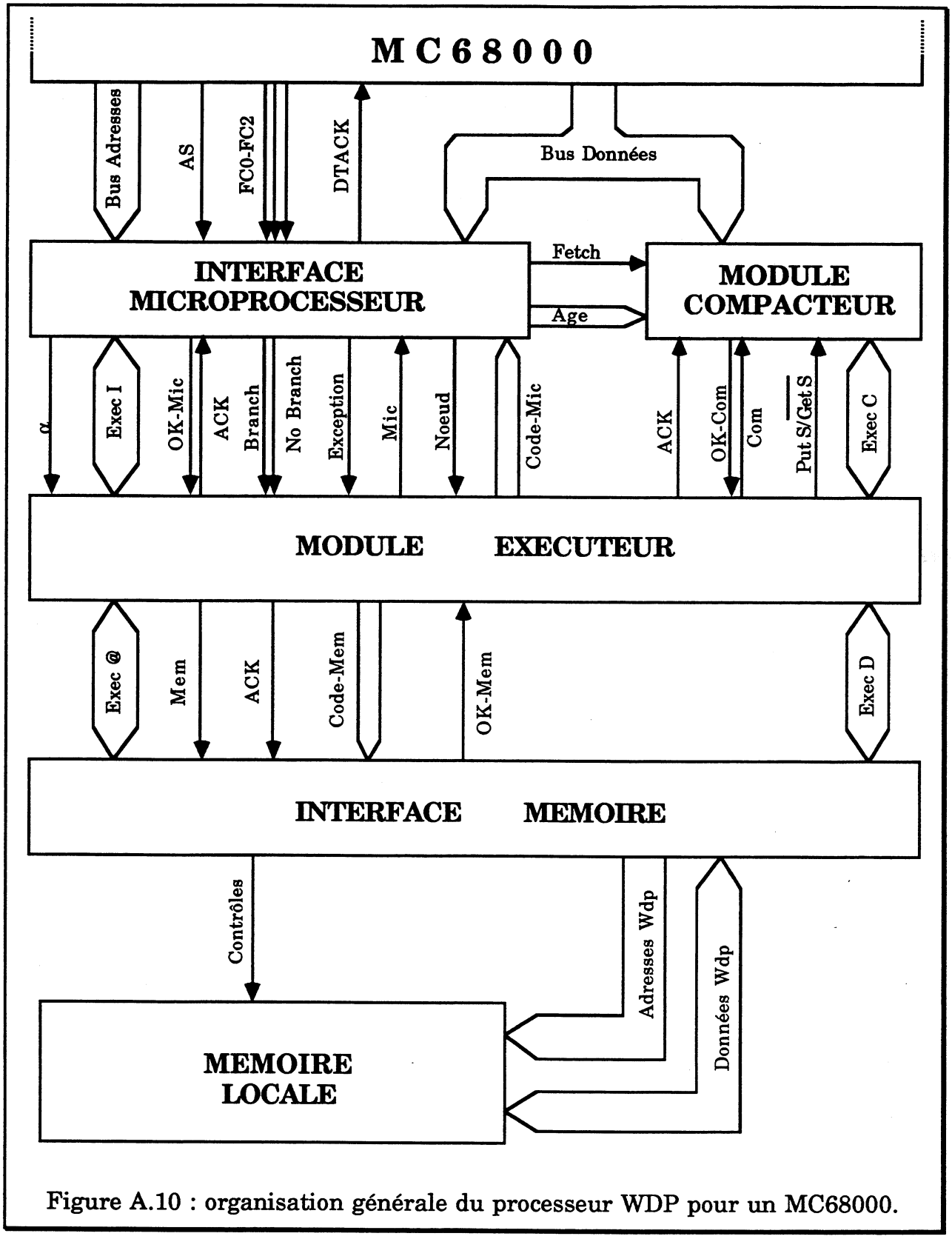


Figure A.10 : organisation générale du processeur WDP pour un MC68000.

Signaux Mic, Com et Mem : ces signaux sont envoyés par la partie exécutrice aux autres modules lorsque celle-ci fait une requête (par exemple, une demande de

signature au module compacteur revient à affirmer Com et à positionner Put S/Get S à 0).

Signaux OK-Mic, OK-Com et OK-Mem : signaux d'acquittement renvoyés à la partie exécutrice par les modules concernés lorsque ceux-ci sont prêts à satisfaire la requête.

Signal ACK : signal envoyé par la partie exécutrice aux autres modules lorsque la requête est satisfaite.

Signal Noeud : généré par l'interface microprocesseur, il avertit la partie exécutrice qu'un noeud a été atteint (c'est-à-dire en cours de chargement ou déjà chargé) par le microprocesseur.

Signaux Branch et No Branch : signaux envoyés par la partie interface microprocesseur pour avertir qu'un branchement a été pris ou non.

Signal α : avertit la partie exécutrice que le noeud en cours de traitement est en cours d'exécution par le microprocesseur (plus simplement, qu'il a atteint l'âge α).

Signal Exception : avertit la partie exécutrice qu'une interruption est survenue (départ en exception détecté).

Ces différents signaux sont utilisés pour la bonne synchronisation des quatre modules. Ils permettent aussi d'avoir un certain niveau d'abstraction pour la partie exécutrice en ce qui concerne le fonctionnement du microprocesseur, et ainsi facilite l'adaptation du processeur watchdog à d'autres microprocesseurs.

3.2.2. Synchronisation des modules

La synchronisation des différents modules ne doit pas entrainer une perte de temps au niveau du traitement. Chacun des modules est réalisé sous la forme PC-PO classique (Partie Contrôle - Partie Opérative). Afin de comprendre les pertes de temps que l'on peut avoir au niveau de la communication entre les parties contrôles des différents modules, étudions un petit exemple.

Supposons que notre système soit constitué de deux modules A et B, chaque module étant formé d'une PC et d'une PO. Les PC communiquent entre elles par l'intermédiaire des signaux En (Enable), Ok et Ack. En et Ack sont générés par la partie contrôle du module A, et Ok est généré par la partie contrôle du module B. Le signal En est utilisé par A pour faire une requête auprès de B qui répond s'il est prêt en affirmant OK. Tant que A n'a pas affirmé Ack, on suppose que B maintient le signal OK positionné (B reste en attente d'acquittement). Les modules A et B fonctionnent sur la même horloge. Si les parties contrôles de A et B changent d'état sur le même front d'horloge, on s'aperçoit (Figure A.11) que le moindre échange

prend deux cycles à A et B (le cycle où Ack est positionné n'est pas un cycle "perdu" pour A puisqu'il peut faire autre chose en même temps).

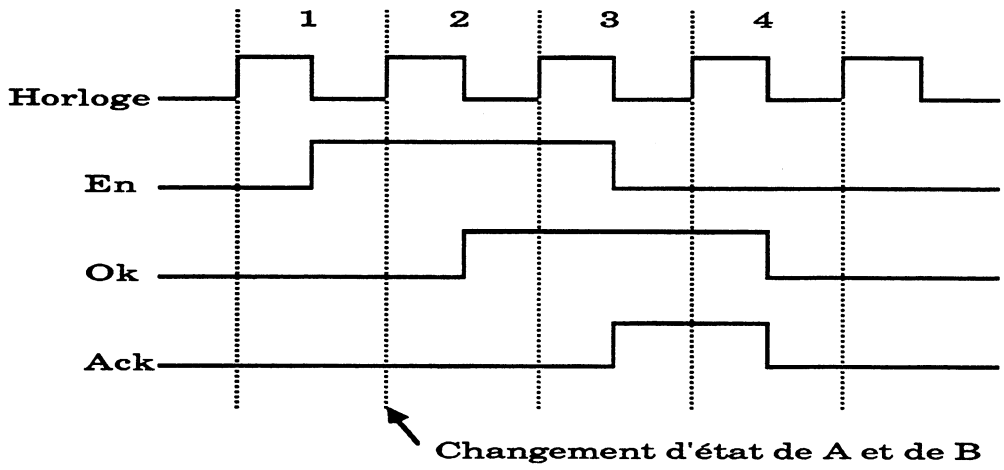


Figure A.11 : synchronisation des modules sur front montant.

Une solution est de synchroniser A sur le front montant de l'horloge, et B sur le front descendant (un peu comme pour une synchronisation PC-PO classique). La partie contrôle de A change donc d'état au front montant de l'horloge, et la partie contrôle de B au front descendant. Les signaux de sortie sont positionnés un demi cycle plus tard (Figure A.12). L'échange peut alors se faire en un cycle, à condition que le retard de sortie des signaux En, Ok et Ack soit suffisamment petit par rapport au cycle d'horloge.

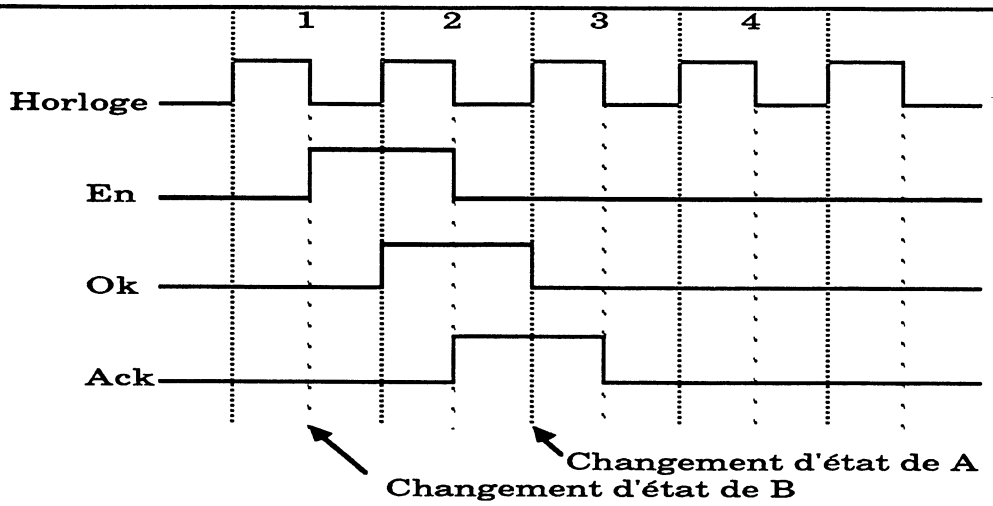


Figure A.12 : synchronisation des modules en opposition de phase.

Cela signifie qu'avec la synchronisation de la Figure A.11, on perd des cycles mais l'horloge peut être très rapide, et avec la synchronisation de la Figure A.12, on ne perd pas de cycle mais l'horloge doit être plus lente ou la logique plus optimisée.

La solution choisie pour le processeur watchdog a été de synchroniser le module exécuteur sur le front montant de l'horloge et les autres modules sur le front descendant (modèle de la Figure A.12). Ceci oblige d'avoir une horloge un peu plus lente, mais les chemins critiques des parties opératives ne permettent pas, de toute façon, d'avoir une horloge plus rapide (soustracteur et incrémenteur au niveau de la PO du module Interface Microprocesseur).

3.3. Description des Modules

La composition de chacun des modules est différente de celle utilisée pour le watchdog du processeur 80386. Au niveau des modules interface microprocesseur et compacteur, ceci est évidemment dû à la prise en compte des exceptions et des pipelines synchrones. La partie opérative du module exécuteur, elle, a été "vidée" au profit de la partie opérative du module interface mémoire, de manière à profiter au mieux du mécanisme de prefetch d'une instruction WDP (fonctionnement parallèle du module interface mémoire et du module exécuteur).

3.3.1. Module Exécuteur

Le module exécuteur est composé d'un registre 4 bits, d'une porte XOR (OU Exclusif) et d'un comparateur "tout à 0".

Le registre est le registre instruction et est appelé RCOP (Registre de Code OPératoire). Son entrée est connectée aux quatre bits de poids fort du bus Exec @. Sa sortie le relie à la partie contrôle. Etant donné que la partie contrôle du module exécuteur est synchronisée sur les fronts montants de l'horloge, la partie opérative est synchronisée sur les fronts descendants. Le registre RCOP récupère la valeur présente sur le bus Exec @ en provenance du module interface mémoire (il mémorise en effet le type de la prochaine instruction watchdog à exécuter). Or la partie opérative de l'interface mémoire est synchronisée sur les fronts montants de l'horloge (et sa partie contrôle sur fronts descendants). Aussi, la valeur présente sur le bus Exec @ n'est pas forcément stable au front descendant de l'horloge, c'est-à-dire au moment où RCOP est censé en lire la valeur. Pour remédier à ce problème, on utilise non pas des bascules sensibles aux fronts pour RCOP, mais des bascules sensibles à niveau, et en particulier ici, sensible au niveau haut.

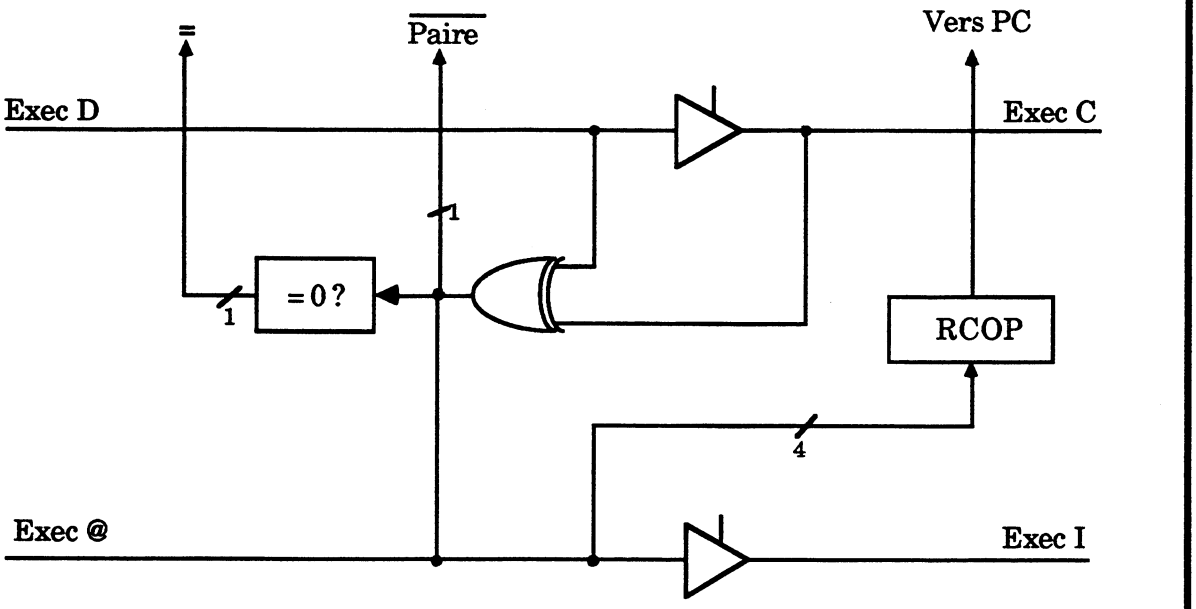


Figure A.13 : partie opérative du module exécuteur (16 bits).

La porte XOR (ou plutôt les 16 portes XOR) a comme entrée le bus Exec D et le bus Exec C. Sa sortie est connectée (par l'intermédiaire d'une porte 3 états), au bus Exec @. Cette (ces) porte est utilisée pour comparer la signature courante en provenance du module compacteur et la signature de référence en provenance du module interface mémoire (lors de l'exécution d'une instruction WDP destination). Le comparateur de "tout à 0" à 16 entrées permet de vérifier que le résultat du OU Exclusif est bien 0. Le XOR est aussi utilisé lors de l'exécution des instructions WDP de séquencement pour effectuer le calcul PCW_{dest} <-- SIGN ⊕ REF(PCW_{actuel}). La signature présente sur le bus Exec C provient du module compacteur, la référence provient du module interface mémoire, et le résultat est déposé sur le bus Exec @ et est destiné à l'interface mémoire.

La partie exécutrice est en fait surtout chargée de diriger les autres modules en fonction des instructions WDP à exécuter et du comportement du microprocesseur sous test, qui lui est rapporté par les signaux du module interface microprocesseur.

3.3.2. Module Interface Mémoire

Le module interface mémoire est chargé d'effectuer les différents échanges avec la mémoire locale. Pour cela il gère les registres PCW et SPW (respectivement compteur ordinal et pointeur de pile), afin d'être le plus indépendant possible du module exécuteur. Il gère le mécanisme de prefetch des instructions watchdog, et entame un cycle mémoire dès qu'un des registres de sa file d'attente de deux instructions WDP (quatre mots) a été libéré (chargement au plus tôt). L'empilement d'information se fait à travers le registre PCW, qui est donc non seulement connecté

au bus d'adresses externe, mais aussi au bus de données (ceci permet d'économiser un registre 16 bits). Un registre MEMO permet de mémoriser la valeur du PCW durant l'exécution de certaines instructions. Tous les registres cités ci-dessus ont une taille de 16 bits.

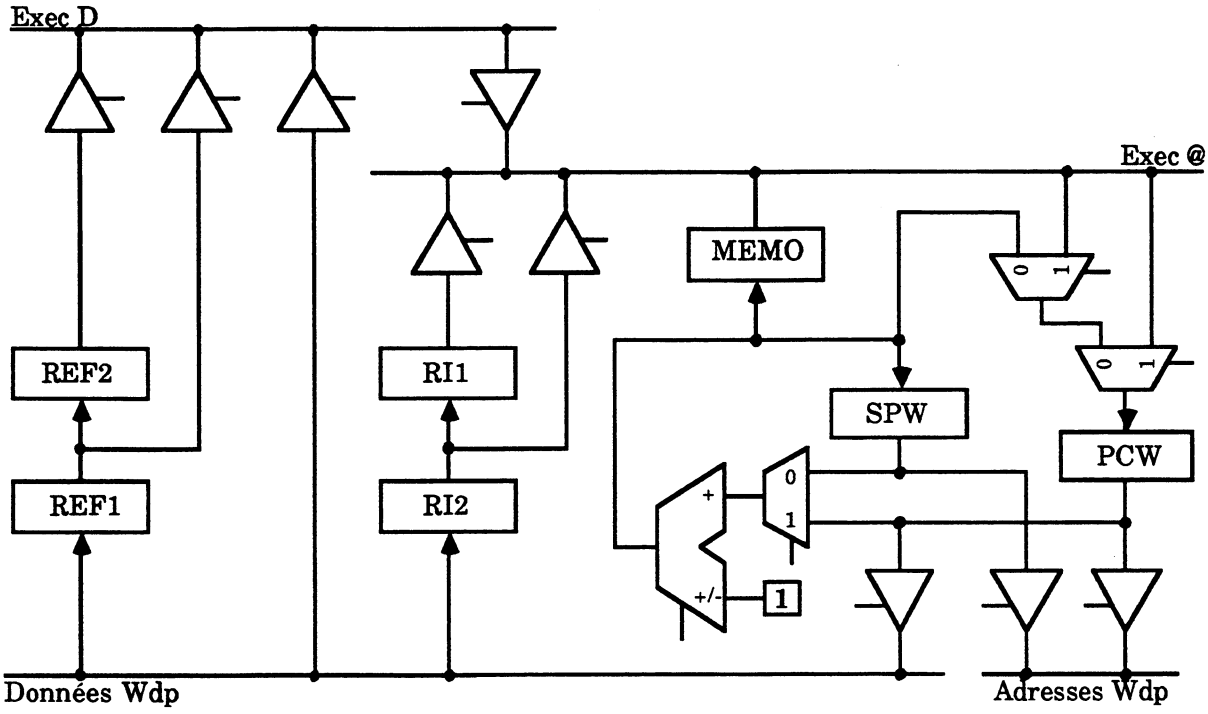


Figure A.14 : partie opérative de l'interface mémoire.

L'intérêt d'avoir un module interface mémoire aussi indépendant de la partie exécutrice (gestion du PCW et du SPW) est :

- De permettre une synchronisation en opposition de phase entre la partie exécutrice et les autres modules, sans pour autant avoir de problèmes de synchronisation, pour les transferts de registres par exemple, entre les différentes parties opératives des modules. En effet, si PCW était géré par la partie exécutrice, le même problème que pour RCOP se poserait dès qu'on voudrait charger une valeur provenant d'un autre module. L'utilisation de bascules sensibles au niveau haut pour PCW ne serait pas une bonne solution. De telles bascules ne pouvant être lues et chargées dans le même cycle, il faudrait deux cycles au lieu d'un pour incrémenter PCW.
- De permettre une gestion plus efficace du mécanisme de prefetch watchdog, du fait du parallélisme de fonctionnement entre la partie exécutrice et l'interface mémoire.

- D'avoir une plus grande indépendance de la partie exécutrice vis à vis de la mémoire utilisée, donc une plus grande souplesse d'implantation au niveau de la carte portant le microprocesseur.

3.3.3. Module Compacteur

Le module compacteur est constitué d'un MISR programmable et d'un registre supplémentaire de manière à stocker la signature le temps que l'instruction microprocesseur correspondante soit exécutée (prise en compte du pipeline synchrone avec traitement des exceptions).

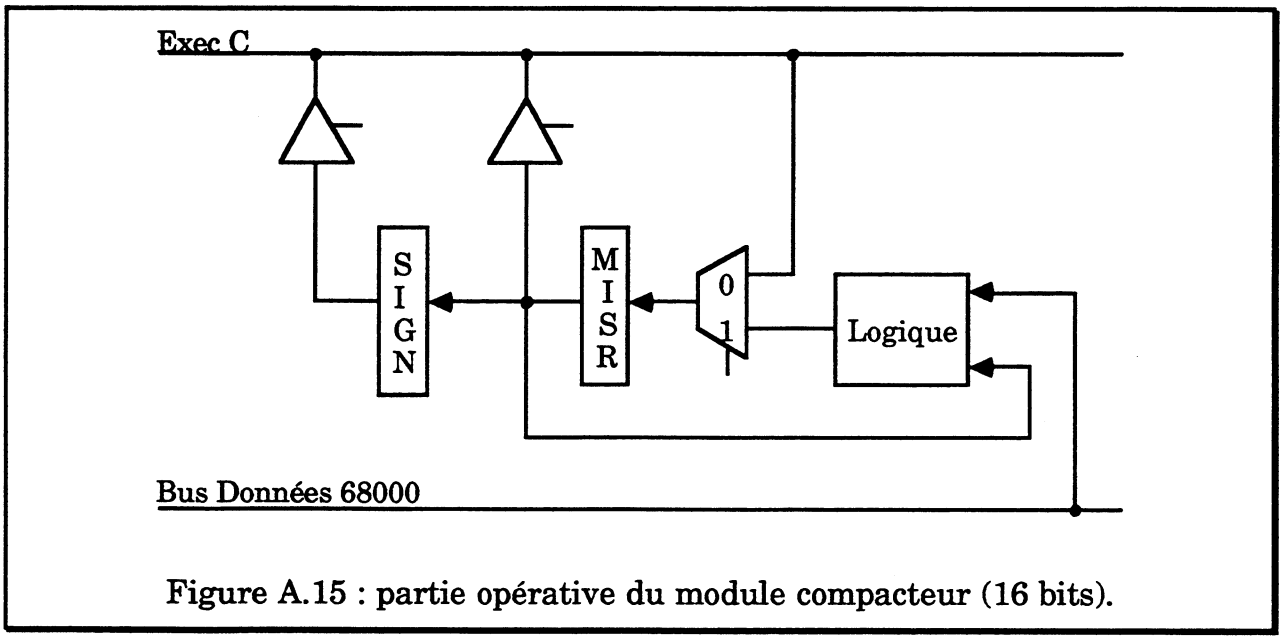


Figure A.15 : partie opérative du module compacteur (16 bits).

3.3.4. Module Interface Microprocesseur

L'interface microprocesseur est le module dont la partie opérative est la plus complexe, du fait en particulier que la taille, en nombre de bits, de ses éléments est assez variable.

L'interface dispose d'un registre sur 23 bits (appelé Aμ), dont le rôle est de stocker l'adresse microprocesseur courante de chargement des instructions. Ce registre est accompagné d'un incrémenteur 23 bits, dont la treizième retenue peut être stockée dans une bascule (pour le compte de la partie contrôle). 23 portes XOR ont comme entrée les 23 bascules du registre Aμ. La deuxième entrée des portes XOR est connectée aux 23 bits du bus d'adresses du 68000. Leur sortie est connectée à un comparateur de "tout à 0" dont la sortie est destinée à la partie contrôle. Ce dispositif sur 23 bits permet de détecter les ruptures de séquence en comparant à chaque fetch, l'adresse microprocesseur réelle de chargement de l'instruction, à l'adresse attendue en séquence.

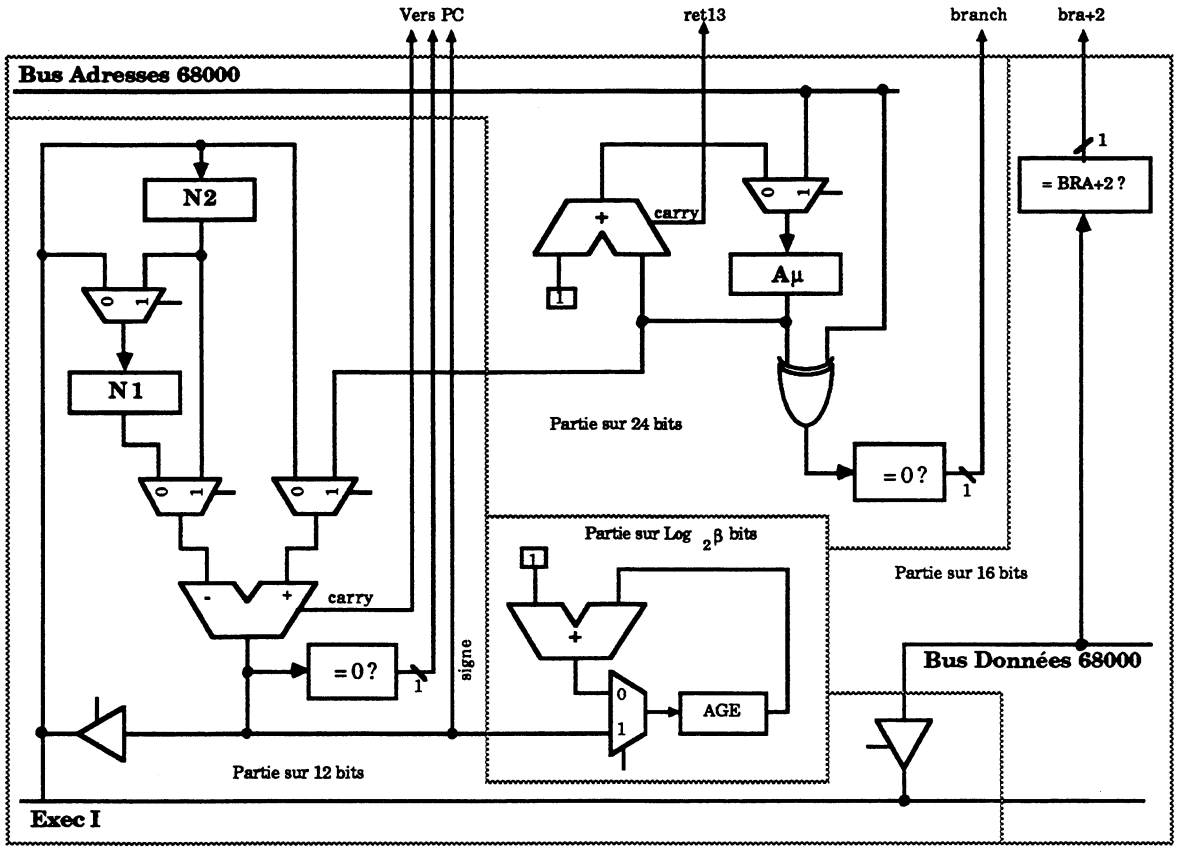


Figure A.16 : partie opérative du module interface micro.

Deux registres sur 12 bits (appelés N1 et N2) permettent de stocker l'adresse microprocesseur du noeud qui vient d'être traitée (ou en cours de traitement), et l'adresse microprocesseur du prochain noeud à traiter. Leur entrée est connectée aux 12 bits de poids faible du bus Exec I. Leur sortie est connectée à un additionneur/soustracteur de manière à pouvoir réaliser l'opération N2-N1. De même, les douze bits de poids faible de la sortie du registre Aμ sont connectés à l'entrée du soustracteur de manière à pouvoir réaliser l'opération Aμ-N2. La sortie du soustracteur est connectée à un comparateur "tout à 0". La retenue sortante est envoyée vers la partie contrôle. Ce dispositif sur 12 bits permet de déterminer si un noeud est atteint en séquence par comparaison de Aμ et de N2.

Un compteur modulo β (ici β=2), est connecté en entrée aux $\log_2(\beta)$ bits de poids faible de la sortie de l'additionneur/soustracteur décrit ci-dessus. La valeur du compteur peut être chargée à partir de cette entrée, ou remise à 0 par un signal. Ce compteur permet de compter le nombre de tops fetch écoulés depuis le chargement, par le microprocesseur, de l'instruction correspondant à un noeud (ce compteur détermine l'âge du noeud). Au moment du fetch du noeud, le compteur est remis à 0. Si le noeud est déjà présent dans le pipeline lorsque le watchdog s'intéresse à lui,

le compteur est réinitialisé avec le résultat de la soustraction Aμ-N2. La valeur du compteur est parfois utilisée pour le choix de la signature du module compacteur et y est donc acheminée.

Enfin, sur les 24 portes XOR décrites précédemment, 16 d'entre elles sont aussi connectées en entrée au bus de données du 68000, et à la valeur du code de l'instruction 68000 "branchement inconditionnel relatif de deux octets". Leur sortie est connectée à un comparateur "tout à 0". Ce dispositif sur 16 bits permet de détecter le passage de l'instruction de branchement attendue après un départ en exception. L'adresse WDP de début du programme de traitement de l'exception est ensuite acheminée du bus de données au bus Exec I (au fetch qui suit).

Ce module d'interface, tel qu'il est conçu, a l'avantage de pouvoir être assez facilement adaptable à différents microprocesseurs (tant que ceux-ci font partie de la catégorie "microprocesseurs à pipeline synchrone"). En principe, il n'y a en effet pas de registres à ajouter. Ce qui change, c'est la taille des différents éléments, à savoir le bloc soustracteur, le bloc Aμ et le bloc compteur. Dans une conception utilisant un compilateur de silicium, la largeur des différents éléments n'est qu'un paramètre de compilation, et est donc simple à changer.

3.4. Exemple d'exécution d'une instruction WDP

Afin de mieux comprendre le fonctionnement du processeur watchdog, nous allons voir l'exemple de l'exécution d'une instruction WDP de type I1, c'est-à-dire destination, du point de vue de la partie exécutrice (cette dernière étant la plus intéressante pour la compréhension du fonctionnement du watchdog puisqu'elle commande les autres).

La Figure A.17 donne une représentation simplifiée du traitement effectué par le contrôleur de la partie exécutrice. Les ellipses représentent des états, les notations en caractères gras le long des arcs représentent les prédicats de transition, et les notations en italique à coté des arcs représentent les commandes envoyées soit aux autres modules, soit à la partie opérative du module exécuteur.

Les signaux **Mic**, **Com**, **Mem**, **Noeud**, **Ack** et α ont été définis en section A.1.3.2 . le signal **Except** a été décrit dans le même paragraphe sous l'appellation "Exception". La notation **Ok** représente la conjonction de plusieurs signaux, à savoir Ok-Mic et Ok-Mem pour l'arc allant de l'état "Attente Noeud" à l'état "Instruction de type I1", et la conjonction des signaux Ok-Mic, Ok-Mem et Ok-Com pour les autres arcs. Ces signaux ont aussi été définis en section A.1.3.2 . La notation = indique que le test de la signature a attesté de la correction de celle-ci, le signe $\neq$ indiquant que la signature est erronée. Code 1 et Code 2 représentent un

ensemble de commandes envoyées aux autres modules et à la partie opératrice. Ils sont expliqués ci-dessous, sans donner le détail des commandes. La signification des différents états est donnée après l'explication de Code 1 et Code 2.

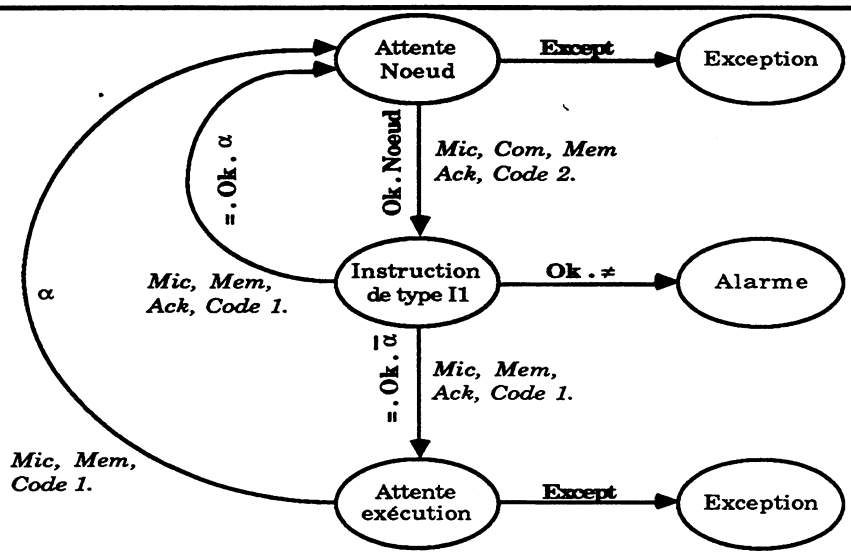


Figure A.17 : exécution d'une instruction de type I1.

Code 1 est une demande de chargement de RCOP. Pour cela, la partie exécutrice demande au module interface mémoire de déposer sur le bus Exec@ la valeur du premier mot de l'instruction WDP suivante (celle qui suit en séquence celle qu'on vient d'exécuter). Elle avertit le module interface microprocesseur que l'adresse du prochain noeud est disponible sur le bus Exec I (connecté au bus Exec @ le temps de la transaction) et qu'elle doit donc la charger dans l'un de ses registres. Enfin, le signal de chargement du registre RCOP est positionné de manière à récupérer le type de l'instruction lors de son passage sur le bus Exec@.

Code 2 correspond à la vérification de la signature. Pour cela, la partie exécutrice demande au module interface mémoire de déposer sur le bus Exec D la valeur du deuxième mot de l'instruction WDP en cours de traitement. Elle demande au module compacteur de déposer la signature courante sur le bus Exec C. Elle demande au module interface microprocesseur de maintenir la valeur du compteur (c'est-à-dire l'âge) stable durant le temps de la transaction, de manière à ce que la valeur de la signature sélectionnée reste stable elle aussi. Ceci permet d'effectuer le XOR (OU Exclusif) entre la valeur courante de la signature et la référence stockée au niveau de l'instruction watchdog et ainsi de tester leur égalité (signature correcte : signe =, signature incorrecte : signe ≠).

Les états "Exception" et "Alarme" correspondent respectivement à un départ en exception et au positionnement du signal d'alarme, une erreur ayant été détectée.

L'état "Attente Noeud" correspond à l'attente du signal Noeud et des signaux Ok-Mic et Ok-Mem. Tant que le noeud suivant n'a pas été atteint et que la demande définie par le Code 1 n'a pas été acquittée, le module exécuteur réitère les signaux Mic, Mem et Code 1.

L'état "Instruction de type I1" correspond à l'attente de l'acquittement de la demande définie par Code 2. Tant que les signaux Ok-Mic, Ok-Com et Ok-Mem ne sont pas tous trois positionnés, le module exécuteur réitère les signaux Mic, Com, Mem et Code 2. Lorsque Ok-Mic, Ok-Com et Ok-mem sont tous trois affirmés, il vérifie que le résultat du XOR est bien 0, c'est-à-dire que la signature courante et la référence du noeud sont bien égales (signe = ou signe ≠).

L'état "Attente exécution" correspond à l'attente de l'exécution, par le microprocesseur, de l'instruction microprocesseur correspondant au noeud destination qui vient d'être traité. Tant que le noeud n'a pas atteint l'âge α, le module exécuteur réitère l'envoi des signaux Mic, Mem et Code 1.

Le graphe donné en Figure A.17 est assez simplifié par rapport au graphe de contrôle réel pour le traitement des instructions destination. Toutefois il a l'intérêt d'illustrer, sans trop entrer dans le détail, le fonctionnement du processeur watchdog. On remarquera que la partie exécutrice synchronise les trois autres modules entre eux par l'attente des trois signaux Ok-Mic, Ok-Mem et Ok-Com, le signal Ack ne les libérant de la requête que lorsque les trois sont réunis. Toutefois, dans la plupart des cas, les trois modules répondent immédiatement. La requête du module exécuteur est alors servie en un seul cycle ce qui signifie que l'instruction WDP peut être exécutée en seulement deux cycles, soit le strict nécessaire.

3.5. Modèle VHDL du watchdog WDP pour MC68000

```
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------  Modele du watchdog WDP  ----------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------

library COMPASS_LIB;
use IEEE.STD_LOGIC_1164.all;
use COMPASS_LIB.COMPASS.all;

entity WATCHD is

        port (  MC_DTACK_IN     :in     bit;
                MC_DTACK_OUT    :out    bit;
                MC_FC           :in     bit_vector(2 downto 0);
                MC_AS           :in     bit;
                MC_ADR          :in     bit_vector(23 downto 1);
                MC_DATA         :in     std_logic_vector(15 downto 0);
                W_ADR           :out    bit_vector(15 downto 0);
                W_DATA          :inout  std_logic_vector(15 downto 0);
                W_DTACK         :in     bit;
                W_AS            :out    bit;
                W_R_Wb          :out    bit;
                ALARME          :out    bit;
                CLK             :in     bit
             );

end WATCHD;


library IEEE;
library COMPASS_LIB;
library WDP_BOARD;

use STD.TEXTIO.all;
use IEEE.STD_LOGIC_1164.all;
use COMPASS_LIB.COMPASS.all;
use WDP_BOARD.UTILITAIRE.all;
use WDP_BOARD.TROIS_ETATS;


architecture A_WATCHD of WATCHD is

------------------------------------------------------------------------------
-------------------------- Definitions  types et sous_types ------------------
------------------------------------------------------------------------------

   type            node_type    is (SEQ,DEST,ANT);

   subtype BV2          is bit_vector(1 downto 0);
   subtype BV3          is bit_vector(2 downto 0);
   subtype nibble       is bit_vector(3 downto 0);
   subtype BV6          is bit_vector(5 downto 0);
   subtype byte         is bit_vector(7 downto 0);
   subtype adr_word     is bit_vector(11 downto 0);
   subtype word         is bit_vector(15 downto 0);
   subtype add_word     is bit_vector(23 downto 1);
   subtype d_word       is bit_vector(31 downto 0);
```

```
------------------------------------------------------------------------------
---------------------- Declaration des signaux -------------------------------
------------------------------------------------------------------------------

------------------------------------------------------------------------------
--- Signaux affectes par le module interface memoire I_MEM
------------------------------------------------------------------------------
-- registres
signal isf_c1          : word;          -- registre de prefetch champ 1
signal isf_c2          : word;          -- registre de prefetch champ 2
       alias   w_ipf   : nibble   is isf_c1(15 downto 12);

-- flags
signal isf1_ok         : boolean;       -- indicateur champ 1 prefetch plein
signal isf2_ok         : boolean;       -- indicateur champ 2 prefetch plein
signal imem_busy       : boolean;       -- indicateur acces bus en cours

-- bus
signal W_DATA_ecr      : word;          -- bus de donnee WDP en ecriture
signal W_DATA_lec      : word;          -- tampon lecture data WDP

-- divers
signal en_W_DATA       : bit;           -- validation ecriture bus WDP
signal ph_bus          : string(1 to 4);       -- phase du bus memoire WDP

------------------------------------------------------------------------------
--- Signaux affectes par le module interface micro
------------------------------------------------------------------------------
-- process IMIC :
------------------------------------------------------------------------------
-- registres
signal seq_fetch       : add_word;      -- adresse du dernier fetch MC68000
signal mc_instr        : word;          -- instruction MC68000 en cours de fetch

-- flags
signal imic_node       : boolean;       -- indicateur de noeud atteint
signal imic_branch     : boolean;       -- indicateur de rupture de
                                        -- sequence MC6800
signal fetch_mc        : boolean;       -- indicateur de fetch MC68000 en cours

------------------------------------------------------------------------------
-- process NODE_AGE_CALC :
------------------------------------------------------------------------------

signal  bloque_mc      : boolean;       -- demande d'arret du micro sur le fetch
                                        -- suivant
signal  node_age       : adr_word;      -- age du noeud present dans next_node
signal  node_passed    : boolean;       -- signale que le noeud dans next_node
                                        -- a ete depase par le micro
signal  node_acces     : boolean;       -- signale que l'acces MC en cours est
                                        -- le noeud courant
```

```
-------------------------------------------------------------------------------
--- Signaux affectes par le module executeur EXEC
-------------------------------------------------------------------------------
-- registres
signal pcw      : word; -- pointeur instruction WDP
signal memo     : word; -- memorisation pointeur instruction WDP
signal spw      : word; -- pointeur de pile WDP
signal ir_c1    : word; -- registre d'instruction champ1
signal ir_c2    : word; -- registre d'instruction champ2

       alias   w_instr   : nibble   is ir_c1(15 downto 12);
       alias   next_node : adr_word is ir_c1(11 downto 0);

-- flags pour I_MEM
signal isf1_read       : boolean;      -- indicateur champ 1 prefetch vide
signal isf2_read       : boolean;      -- indicateur champ 2 prefetch vide
signal empil_pcw       : boolean;      -- demande d'empilememt du PCW
signal empil_sign      : boolean;      -- demande d'empilememt de la signature
signal depil           : boolean;      -- demande de depilememt

-- flags pour I_MIC
signal node_ok  : boolean;      -- si FALSE -> arret MC6800 pendant fetch

-- flags pour NODE_AGE_CALC
signal  node_charged    : boolean;      -- indique aue le registre d'instruction
                                        -- ir_cx a ete modifie

-- flags pour COMPAC
signal s_init : boolean;        -- demande d'initialisation de signature
signal s_load : boolean;        -- demande de chargement de la signature


-------------------------------------------------------------------------------
--- Signaux affectes par le module compacteur COMPAC
-------------------------------------------------------------------------------
-- registres
signal signature        : word; -- registre de signature a I
signal sign_decal       : word; -- registre de signature a I - 1
-- flags

-------------------------------------------------------------------------------
---------------------- definition des constantes  -----------------------------
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- signification du mot d'etat FC2,FC1,FC0 du MC68000
-------------------------------------------------------------------------------
constant CODE_USER      :BV3:="010";
constant CODE_SU        :BV3:="110";
constant INT_ACK        :BV3:="111";
```

```
-------------------------------------------------------------------------------
-- definition des instructions WDP
-------------------------------------------------------------------------------
constant I0:nibble := "0000";
constant I1:nibble := "0001";
constant I2:nibble := "0010";
constant I3:nibble := "0011";
constant I4:nibble := "0100";
constant I5:nibble := "0101";
constant I6:nibble := "0110";
constant I7:nibble := "0111";
constant I8:nibble := "1000";
constant I9:nibble := "1001";

constant AUCUNE:nibble := "1111";

-------------------------------------------------------------------------------
-- divers
-------------------------------------------------------------------------------

constant PERVERS_BRA : word := x"6002";

constant BV23_1                 : add_word := "00000000000000000000001";

-- nombre de T_wait pour les acces memoire
constant N_TWAIT        : integer       := 3;

-- age d'une instruction micro  au moment de son execution
constant BETA           : adr_word      := x"002";

-------------------------------------------------------------------------------
-- division polynomiale: polynome diviseur
-------------------------------------------------------------------------------
constant POL:word:=x"8811";

-------------------------------------------------------------------------------
----------declaration de composant portes 3 etats ----------------------------
-------------------------------------------------------------------------------

component WDP_TS
        generic(NB      :integer);
        port(   ENTREE  :in     bit_vector(NB-1 downto 0);
                SORTIE  :out    std_logic_vector(NB-1 downto 0);
                ENABLE  :in     bit
                );
end component;

for all:WDP_TS use entity TROIS_ETATS(A_TROIS_ETATS);
```

```
------------------------------------------------------------------------------
-------------------- debut instructions architecture A_WATCH -----------------
------------------------------------------------------------------------------

begin

------------------------------------------------------------------------------
-------------------- Instanciation des composants ----------------------------
------------------------------------------------------------------------------

TS1:WDP_TS
        generic map(16)
        port map(W_DATA_ecr,W_DATA,en_W_DATA);


------------------------------------------------------------------------------
-------------------- Module Interface Microprocesseur MI ---------------------
------------------------------------------------------------------------------

IMIC:process

-- synchronisation des signaux flags :
--      fetch_mc        : CLK = 1       = synchro mormale
--      imic_node       : CLK = 0       --> pour gagner 1/2 cycle
--      imic_branch     : CLK = 0       --> pour gagner 1/2 cycle

begin


-- attente demarage cycle bus MC68000
while MC_AS /= '0' loop
        MC_DTACK_OUT    <= '1';
        fetch_mc        <= FALSE;
        imic_branch     <= FALSE;
        imic_node       <= FALSE;
        wait on CLK until CLK = '0';
end loop;

if  (MC_FC = CODE_USER) or (MC_FC = CODE_SU) then
        seq_fetch <= seq_fetch + BV23_1;

        if MC_ADR = seq_fetch then
                imic_branch <= FALSE;
        else    imic_branch <= TRUE;
                seq_fetch   <= MC_ADR + BV23_1;
        end if;

        if not node_ok then
                wait on node_ok until node_ok;
        end if;

        if MC_ADR(12 downto 1) = next_node then
                imic_node <= TRUE;
        else    imic_node <= FALSE;
        end if;

        if MC_DTACK_IN = '1' then
                wait on MC_DTACK_IN until MC_DTACK_IN = '0';
        end if;

        if not node_ok then
                wait on node_ok until node_ok;
        end if;
```

```
        MC_DTACK_OUT    <= '0';
        wait on CLK until CLK = '1'; --> valide ram68 avant lecture MC_DATA

        mc_instr        <= to_bit(MC_DATA);
        fetch_mc        <= TRUE;


else    -- acces memoire hors code

        if MC_DTACK_IN = '1' then
                wait on MC_DTACK_IN until MC_DTACK_IN = '0';
        end if;

        MC_DTACK_OUT    <= '0';


end if;

assert (MC_AS = '0')
        report " RAM68 : violation temps acces"
        severity FAILURE;

wait on CLK until CLK = '1';

end process IMIC;

-------------------------------------------------------------------------------
-- process de calcul de l'age d'un noeud
-- memorise quamd un noeud vient d'etre atteint
-- bloque le micro quand :
--      - l'age du noeud courant est > 1
-------------------------------------------------------------------------------
CALCUL_AGE_NODE: process
begin

node_age        <= x"000";
node_passed     <= FALSE;
bloque_mc       <= FALSE;
node_acces      <= FALSE;

wait on node_charged until node_charged;

loop

wait on CLK until CLK = '1';

if imic_node and not node_acces then
        node_acces      <= TRUE;
        node_passed     <= TRUE;
        node_age        <= x"000";
end if;

if not imic_node then
        node_acces      <= FALSE;
end if;

if fetch_mc and node_passed then
        node_age        <= node_age + x"001";
end if;
```

```
if node_charged then
        if MC_ADR(12 downto 1) < next_node then
                node_passed     <= FALSE;
                node_age        <= x"000";
        else
                node_age        <= MC_ADR(12 downto 1) - next_node;
                node_passed     <= TRUE;
        end if;
end if;

if node_age >= x"002" then
        bloque_mc       <= TRUE;
else    bloque_mc       <= FALSE;
end if;

end loop;

end process CALCUL_AGE_NODE;

-------------------------------------------------------------------------------
----------------------- Module Interface Memoire ------------------------------
-------------------------------------------------------------------------------

IMEM:process

-------------------------------------------------------------------------------
-- lecture memoire : nombre de cycles defini par la constante N_TWAIT
-------------------------------------------------------------------------------
procedure read_mem(adresse : in word;signal read_value : out word) is
variable        wait_cptr       : integer := 0;
begin
        imem_busy       <= TRUE;
        en_W_DATA       <= '0';
        W_AS            <= '1';
        W_R_Wb          <= '1';
        W_ADR           <= adresse;
        ph_bus          <= "PH 0";
        W_AS            <= '0';
        wait on CLK until CLK = '0';

        while wait_cptr /= N_TWAIT loop
                ph_bus  <= "WAIT";
                wait_cptr := wait_cptr + 1;
                wait on CLK until CLK = '0';
        end loop;

        read_value      <= to_bit(W_DATA);
        W_AS            <= '1';
        wait on CLK until CLK = '1';

        ph_bus          <= "PH Z";

end read_mem;
```

```
-------------------------------------------------------------------------------
-- ecriture memoire : nombre de cycles defini par la constante N_TWAIT
-------------------------------------------------------------------------------
procedure write_mem(adresse : in word;signal written_value : in word) is
variable        wait_cptr       : integer := 0;
begin
        imem_busy       <= TRUE;
        en_W_DATA       <= '1';
        W_AS            <= '1';
        W_R_Wb          <= '0';
        W_ADR           <= adresse;
        W_DATA_ecr      <= written_value;
        ph_bus          <= "PH 0";
        W_AS            <= '0';
        wait on CLK until CLK = '0';

        while wait_cptr /= N_TWAIT loop
                ph_bus  <= "WAIT";
                wait_cptr := wait_cptr + 1;
                wait on CLK until CLK = '0';
        end loop;

        W_AS            <= '1';
        wait on CLK until CLK = '1';

        ph_bus          <= "PH Z";
        en_W_DATA       <= '0';
        W_R_Wb          <= '1';

end write_mem;

begin
-- initialisations

        W_AS            <= '1';
        W_R_Wb          <= '1';
        isf1_ok         <= FALSE;
        isf2_ok         <= FALSE;
        imem_busy       <= FALSE;

-- attent la premiere demande de charger_noeud_suiv

while not (isf1_read or isf2_read) loop
        wait on CLK until CLK = '1';
end loop;

-- attent isf1_read ou isf2_read  = registre de prefetch vide
-- pour declencher le chargement du
-- registre de prefetch isf_c1 ou isf_c2 correspondant
-- renvoie isf1_ok et isf2_ok lorsque le registre de prefetch
-- correspondant est plein

loop

        wait on CLK until CLK = '1';

        if isf1_read then
                isf1_ok <= FALSE;
        end if;

        if isf2_read then
                isf2_ok <= FALSE;
        end if;
```

```
        if not isf1_ok then
                read_mem(pcw,isf_c1);
                isf1_ok <= TRUE;
        end if;

        if not isf2_ok then
                read_mem(pcw + x"0001",isf_c2);
                isf2_ok <= TRUE;
        end if;

        if empil_pcw then
                write_mem(spw,memo);
        end if;

        if depil then
                read_mem(spw + x"0001",W_DATA_lec);
        end if;

        if empil_sign then
                write_mem(spw,sign_decal);
        end if;

        imem_busy       <= FALSE;


end loop;
end process IMEM;

-------------------------------------------------------------------------------
-------------------------- Module Executeur EXEC ------------------------------
-------------------------------------------------------------------------------

EXEC:process

procedure charger_noeud_suiv(node_req : in node_type);
procedure exec_node;
procedure exec_I0;
procedure exec_I1;
procedure exec_I2;
procedure exec_I3;
procedure exec_I4;
procedure exec_I5;
procedure exec_I6;
procedure exec_I7;
procedure exec_I8;
procedure exec_I9;
procedure unexpected_branch(type_I : in nibble;age : in adr_word);
procedure exception_ack;

-----------------------------------------------------------

procedure wait_sync(cond:boolean) is
begin

wait on CLK until CLK = '1';

while not cond loop
       wait on CLK until CLK = '1';
end loop;

end wait_sync;
```

```
------------------------------------------------------------

procedure active(signal commande : out boolean) is
begin
        commande        <= TRUE;
        wait on CLK until CLK = '1';
        commande        <= FALSE;
end active;

------------------------------------------------------------

procedure prefetch_WDP is
begin
-- il faut que IMEM soit libre pour que la demande soit enregistree
        if imem_busy then wait_sync(not imem_busy);
        end if;

-- demande de prefetch du noeud suivant (isfX_read <= TRUE)
        isf1_read       <= TRUE;
        isf2_read       <= TRUE;
        wait on CLK until CLK = '1';

-- suppression de la demande de prefetch
        isf1_read       <= FALSE;
        isf2_read       <= FALSE;

end prefetch_WDP;


------------------------------------------------------------
-- Charge ir_c1 et ir_c2 a partir de isf_c1 et isf_c2
-- attent que les 2 mots de l'instruction soient charges
-- avant de rendre la main
-- node_ok est renvoye = TRUE
-- Optimisation prefetch : eviter le prefetch du noeud
-- en sequence apres un branchement inconditionnel.
-- Problème : si optimisation memoire on peut avoir
-- un noeud destination de type branchement inconditionnel
-- et dans ce cas il faut charger le noeud en sequence.
--> differenciation des noeuds charges (node_req = SEQ,DEST)
------------------------------------------------------------
procedure charger_noeud_suiv(node_req : in node_type) is
begin

if not(isf1_ok and isf2_ok) then
        node_ok         <= FALSE;
        wait_sync(isf1_ok and isf2_ok);
end if;

ir_c1           <= isf_c1;
ir_c2           <= isf_c2;
node_charged    <= TRUE;

if (node_req /= DEST) and (
        (w_ipf = I2) or
        (w_ipf = I4) or
        (w_ipf = I5) or
        (w_ipf = I6) or
        (w_ipf = I9) ) then
-- pas de demande de prefetch (branchement inconditionnel en sequence)
wait on CLK until CLK = '1';

else
```

```
-- demande de prefetch (autes noeuds)
-- increment du PCW
        pcw             <= pcw + x"0002";
        prefetch_WDP;
end if;
node_ok                 <= TRUE;
node_charged    <= FALSE;

end charger_noeud_suiv;

-------------------------------------------------------------
-- execution d'une instruction WDP
-------------------------------------------------------------
procedure exec_node is

begin

        case w_instr is
                when I0 => exec_I0;
                when I1 => exec_I1;
                when I2 => exec_I2;
                when I3 => exec_I3;
                when I4 => exec_I4;
                when I5 => exec_I5;
                when I6 => exec_I6;
                when I7 => exec_I7;
                when I8 => exec_I8;
                when I9 => exec_I9;

                when others => null;

        end case;
end exec_node;

-------------------------------------------------------------
-- noeud de type I0, initialisation
-------------------------------------------------------------
procedure exec_I0 is

begin

active(s_init);
charger_noeud_suiv(SEQ);

end exec_I0;

-------------------------------------------------------------
-- noeud de type I1, destination
-------------------------------------------------------------
procedure exec_I1 is

begin

if signature /= ir_c2 then
        ALARME  <= '1';
end if;

charger_noeud_suiv(SEQ);

end exec_I1;
```

```
------------------------------------------------------------
-- noeud de type I2, branchement inconditionnel
------------------------------------------------------------
procedure exec_I2 is
begin

wait_sync(fetch_mc);

-- signature valide 1 cycle apres
wait on CLK until CLK = '1';            -- 1 cycle apres fetch MC

-- hachage inverse
memo    <= pcw;
pcw     <=  signature xor ir_c2;
prefetch_WDP;                           -- 2 cycles apres fecth MC

wait_sync(fetch_mc or imic_branch);     -- prefetch micro
if imic_branch then
        pcw     <= memo;                -- retour  d'exception sur noeud
        unexpected_branch(w_instr,node_age);
else
        wait_sync(fetch_mc or imic_branch);
        if not imic_branch then
                ALARME  <= '1';
        end if;

        charger_noeud_suiv(DEST);
        if MC_ADR(12 downto 1) /= next_node then
                active(s_init);         -- retour d'exception sur dest
                unexpected_branch(I2,BETA);
        else
                active(s_init);
                charger_noeud_suiv(SEQ);
        end if;
end if;

end exec_I2;

------------------------------------------------------------
-- noeud de type I3, branchement conditionnel
------------------------------------------------------------
procedure exec_I3 is

begin
wait_sync(fetch_mc);
-- signature valide 1 cycle apres
wait on CLK until CLK = '1';

-- hachage inverse
memo    <= pcw;         -- pointe un noeud apres le branchement
pcw     <=  (signature xor ir_c2) - x"0002";
        -- le PCW est decremente de 2 pour compenser l'increment de 2 lors de
        -- l'appel de prefetch par charger_noeud_suiv

wait on CLK until CLK = '1';

charger_noeud_suiv(DEST);
        -- on charge le noeud suivant le BXX dans IR
        -- et on lance le prefetch du noeud destination (nouveau PCW)

wait_sync(fetch_mc or imic_branch);     -- prefetch micro
```

```
if imic_branch then
    pcw         <= memo;                -- retour d'exception sur noeud
    unexpected_branch(w_instr,node_age);
else
    wait_sync(fetch_mc or imic_branch or imic_node);

    if imic_branch then
        charger_noeud_suiv(DEST);
        if MC_ADR(12 downto 1) /= next_node then
                pcw     <= memo - x"0002";
                -- on considere que le branch n'est pas
                -- pris --> retour sur noeud
                unexpected_branch(I3,BETA);
        else
                active(s_init);
                charger_noeud_suiv(SEQ);
        end if;


    else
        -- micro continue en sequence = branch non pris et pas d'exception
        -- le noeud suivant BXX est deja charge dans IR
        -- mais le prefetch contient le noeud destination
        --> il faut lancer le prefetch du noeud en sequence
        -- seulement si le type du noeud suivant n'est pas
        -- une rupture de sequence inconditionnelle
        if ((w_instr = I2) or
            (w_instr = I4) or
            (w_instr = I5) or
            (w_instr = I6) or
            (w_instr = I9) ) then
         -- pas de demande de prefetch
                pcw             <= memo;

        else    if not(isf1_ok and isf2_ok) then
                        node_ok         <= FALSE;
                        wait_sync(isf1_ok and isf2_ok);
        -- il faut attendre la fin du prefetch du noeud destination
        -- ce qui fait perdre du temps inutilement car il ne sera
        -- pas utilisee (branch non pris)
                end if;
                pcw             <= memo + x"0002";
                node_ok         <= TRUE;
                prefetch_WDP;
        end if;

        wait on CLK until CLK = '1';    -- pour attendre isfx_ok = FALSE
        if imic_node then               -- noeud 1 fetch apres branchement
                exec_node;              -- conditionnel non pris
        end if;

    end if;
end if;

end exec_I3;
```

```
------------------------------------------------------------
-- noeud de type I4, depart inconditionnel en ss_programme
------------------------------------------------------------
procedure exec_I4 is

begin
wait_sync(fetch_mc);
wait on CLK until CLK = '1';
-- hachage inverse
memo    <= pcw;
pcw     <=  signature xor ir_c2;

if imem_busy then
        node_ok <= FALSE;
        wait_sync(not imem_busy);
        node_ok <= TRUE;
end if;
-- imem_busy = FALSE
active(empil_pcw);

wait_sync(fetch_mc or imic_branch);     -- prefetch micro

if imic_branch then
    pcw                 <= memo;                -- retour d'exception sur noeud
    unexpected_branch(w_instr,node_age);

else
-- on lance le prefetch du noeud destination (nouveau PCW)
    node_ok     <= FALSE;
    prefetch_WDP;
    node_ok     <= TRUE;
    spw         <= spw - x"0001";               -- le depart est execute

    wait_sync(fetch_mc or imic_branch);
    if not imic_branch then
        ALARME  <= '1';
    end if;

    charger_noeud_suiv(DEST);
    if MC_ADR(12 downto 1) /= next_node then
        active(s_init);                 -- retour d'exception sur dest
        unexpected_branch(I4,BETA);
    else
        active(s_init);
        charger_noeud_suiv(SEQ);
    end if;
end if;

end exec_I4;
```

```
------------------------------------------------------------
-- noeud de type I5, retour inconditionnel de ss_programme
------------------------------------------------------------
procedure exec_I5 is

begin
wait_sync(fetch_mc);            --> signature valide 1 cycle apres
wait on CLK until CLK = '1';

-- test de signature :
if signature /= ir_c2 then
        ALARME  <= '1';
end if;

-- depilement du PC :
if imem_busy then
        node_ok <= FALSE;
        wait_sync(not imem_busy);
        node_ok <= TRUE;
end if;
-- imem_busy = FALSE
active(depil);

wait_sync(fetch_mc or imic_branch);     -- prefetch micro

if imic_branch then
    unexpected_branch(w_instr,node_age);
else
    -- chargement du pcw de retour :
    node_ok     <= FALSE;
    wait_sync(not imem_busy);
    pcw         <= W_DATA_lec;
    prefetch_WDP;
    node_ok     <= TRUE;
    spw         <= spw + x"0001";       -- le retour est execute

    wait_sync(fetch_mc or imic_branch);

    if not imic_branch then
        ALARME  <= '1';
    end if;

    charger_noeud_suiv(DEST);
    if MC_ADR(12 downto 1) /= next_node + x"001" then
        active(s_init);         -- retour d'exception sur dest (depart sspgm)
        unexpected_branch(I5,BETA);
    else
        active(s_init);
        charger_noeud_suiv(SEQ);
    end if;
end if;

end exec_I5;
```

```
------------------------------------------------------------
-- noeud de type I6 , retour inconditionnel d'exception
------------------------------------------------------------
procedure exec_I6 is

begin
wait_sync(fetch_mc);            --> signature valide 1 cycle apres
wait on CLK until CLK = '1';

-- test de signature :
if signature /= ir_c2 then
       ALARME  <= '1';
end if;

-- depilement du PC :
if imem_busy then
       node_ok <= FALSE;
       wait_sync(not imem_busy);
       node_ok <= TRUE;
end if;
-- imem_busy = FALSE
active(depil);

wait_sync(fetch_mc or imic_branch);     -- prefetch micro

if imic_branch then
    unexpected_branch(w_instr,node_age);

else
    node_ok     <= FALSE;
    -- chargement du pcw de retour :
    wait_sync(not imem_busy);
    spw                 <= spw + x"0001";
    pcw                 <= W_DATA_lec;

    -- depilement de la signature :
    active(depil);
    wait_sync(not imem_busy);
    spw                 <= spw + x"0001";
    active(s_load);

    -- chargement du mot de controle :
    active(depil);
    wait_sync(not imem_busy);
    spw                 <= spw + x"0001";
    memo        <= W_DATA_lec;

    prefetch_WDP;
    wait on CLK until CLK = '1';        -- pour que isfX_ok = FALSE

    charger_noeud_suiv(SEQ);            -- renvoie node_ok = TRUE

    -- imic_node et imic_branch arrivent en meme temps et avant fetch_mc
    if not imic_branch  then
        wait_sync(fetch_mc or imic_branch);
    end if;

    if not imic_branch then
       ALARME  <= '1';
    end if;
```

```
        -- test du mot de controle :
        if memo(15 downto 12) = I3 and memo(11 downto 0) = BETA then
        -- depart en exception sur le branchement  -> condition inconue
        --> noeud depile = branchement conditionnel (deja passe)
              if MC_ADR(12 downto 1) = next_node +  x"001" then
              -- branchement non pris = retour sur noeud + 1
                    node_ok <= FALSE;
                    wait on CLK until CLK = '1';
                    charger_noeud_suiv(SEQ);
              else
              -- branchement pris
                    node_ok <= FALSE;
                    pcw     <= (signature xor ir_c2);
              -- on modifie le pcw pendant le prefetch (avant le premier acces) !!!
                    wait on CLK until CLK = '1';
                    charger_noeud_suiv(DEST);
                    if MC_ADR(12 downto 1) /= next_node then
                            ALARME  <= '1';
                    else
                            active(s_init);
                            charger_noeud_suiv(SEQ);
                    end if;
              end if;
        end if;

        wait_sync(fetch_mc or imic_node);
        if imic_node then
           exec_node;              -- retour sur un noeud
        end if;

end if;

end exec_I6;

-------------------------------------------------------
-- noeud de type I7, depart conditionnel en ss_programme
-------------------------------------------------------
procedure exec_I7 is

begin

end exec_I7;

-------------------------------------------------------
-- noeud de type I8, retour conditionnel de ss_programme
-------------------------------------------------------
procedure exec_I8 is

begin

end exec_I8;

-------------------------------------------------------
-- noeud de type I9, depart inconditionnel en exception
-------------------------------------------------------
procedure exec_I9 is

begin
end exec_I9;
```

```
-------------------------------------------------------
-- noeud de type I10, depart conditionnel en exception
-------------------------------------------------------
procedure exec_I10 is

begin
end exec_I10;

-------------------------------------------------------
-- noeud de type I11, retour conditionnel d'exception
-------------------------------------------------------
procedure exec_I11 is

begin
end exec_I11;

-------------------------------------------------------
-- noeud de type I12, illegal
-------------------------------------------------------
procedure exec_I12 is

begin
end exec_I12;

-------------------------------------------------------
-- noeud de type I13, illegal
-------------------------------------------------------
procedure exec_I13 is

begin
end exec_I13;

-------------------------------------------------------
-- noeud de type I14, illegal
-------------------------------------------------------
procedure exec_I14 is

begin
end exec_I14;

-------------------------------------------------------
-- noeud de type I15, illegal
-------------------------------------------------------
procedure exec_I15 is

begin
end exec_I15;


-------------------------------------------------------
-- traitement branchements inoppines
-------------------------------------------------------
procedure unexpected_branch(type_I : in nibble;age : in adr_word) is
begin
node_ok                 <= FALSE;
wait on CLK until CLK = '1';    -- si demande de prefetch -> imem_busy = TRUE

-- empilement du mot de controle
memo(15 downto 12)      <= type_I;
memo(11 downto 0)       <= age;
wait_sync(not imem_busy);
empil_pcw       <= TRUE;
wait_sync(imem_busy);
```

```
empil_pcw       <= FALSE;
spw             <= spw - x"0001";

-- empilement de la signature
wait_sync(not imem_busy);
empil_sign      <=  TRUE;
wait_sync(imem_busy);
empil_sign      <= FALSE;
spw             <= spw - x"0001";

-- empilement du PCW
memo            <= pcw;         -- on empile toujours memo et pas pcw
wait_sync(not imem_busy);
empil_pcw       <= TRUE;
wait_sync(imem_busy);
empil_pcw       <= FALSE;
spw             <= spw - x"0001";
wait_sync(not imem_busy);

exception_ack;

end unexpected_branch;

-------------------------------------------------------
-- traitement des exceptions
-------------------------------------------------------
procedure exception_ack is
begin
-- le PCW, le mot de controle et la signature sont deja empiles
-- le fetch de la destination du branchement innopine peut avoir eu lieu
-- ou non -> tester mc_instr = PERVERS_BRA

node_ok                 <= TRUE;
if mc_instr = PERVERS_BRA then
        null;
else
        wait_sync(fetch_mc);
end if;

if mc_instr /= PERVERS_BRA then
        ALARME  <= '1';
else
        wait_sync(fetch_mc);
        pcw     <= mc_instr;

-- le pcw vient de changer -> demande de chargement de l'instruction
        prefetch_WDP;
        wait on CLK until CLK = '1';     -- pour attendre que isfx_ok = FALSE

-- le prefetch est lance -> demande de chargement de l'instruction
        charger_noeud_suiv(SEQ);         -- renvoie node_ok = TRUE

        if w_instr = I0 then
                if MC_ADR(12 downto 1) /= next_node + x"001" then
                        ALARME  <= '1';
                end if;
                exec_I0;
        else
                ALARME  <= '1';
        end if;
end if;

end exception_ack;
```

```
------------------------------------------------------------------------------
--- debut sequentiel process EXEC
------------------------------------------------------------------------------

begin

-- initialisation
node_ok <= TRUE;                -- pour ne pas bloquer MC68000

isf1_read       <= FALSE;       -- pour ne pas lancer de fetch
isf2_read       <= FALSE;

s_init          <= FALSE;
ALARME          <= '0';

spw             <= x"1FFF";     -- la pile descend

-- reconnaissance pattern interruption
while mc_instr /= PERVERS_BRA loop
        wait_sync(fetch_mc);
end loop;

wait_sync(fetch_mc);

pcw     <= mc_instr;

-- le pcw vient de changer -> demande de chargement de l'instruction
prefetch_WDP;
wait on CLK until CLK = '1';    -- pour attendre que isfx_ok = FALSE

-- le prefetch est lance -> demande de chargement de l'instruction
charger_noeud_suiv(SEQ);        -- renvoie node_ok = TRUE

if w_instr = I0 then exec_I0;
end if;

loop
-- node_ok = TRUE car positionne par charger_noeud_suiv

wait_sync(imic_node or imic_branch);

if imic_branch then
        -- rupture de sequence MC68000 inopinee :
        -- le pcw n'a pas ete incremente par l'appel charger_noeud_suiv
        -- si le noeud courant est un branchement inconditionnel
        if w_instr = I2 or
           w_instr = I4 or
           w_instr = I5 or
           w_instr = I9 then
                null;
        else
                pcw     <= pcw - x"0002";
        end if;
        unexpected_branch(w_instr,node_age);
else
        exec_node;              -- occurence d'un noeud
end if;

-- le noeud suivant est charge et le micro est libere (node_OK = TRUE)
-- car toutes les instructions font un charger_noeud_suiv avant de rendre
-- la main
```

```
-- si le noeud a ete traite avant la fin du cycle de lecture Micro
-- correspondand au noeud, il faut attendre la fin de ce cycle

if node_acces then
        wait_sync(not node_acces);
end if;


-- pour les noeuds qui necessite un empilement ou un depilement du PCW
-- il est possible de le faire avant que le noeud soit atteint (a faire)

end loop;


end process EXEC;


-------------------------------------------------------------------------------
---------------------------- Module Compacteur --------------------------------
-------------------------------------------------------------------------------


COMPAC:process

-----------------------------------------------------------

procedure compacter is

variable in_misr: word;

begin

sign_decal      <= signature;

for J in 15 downto 1 loop
        in_misr(J) := signature(J-1)
                        xor (POL(J) and signature(15))
                        xor mc_instr(J);
end loop;

in_misr(0) := (POL(0) and signature(15)) xor mc_instr(0);

signature <= in_misr;

end compacter;

-----------------------------------------------------------

begin

wait on CLK until CLK = '1';

-- chargement de la signature apres depilement
if s_load then
        signature       <= W_DATA_lec;
end if;

-- initialisation de la signature sur noeud destination
if s_init then
        signature       <= ir_c2;
        wait on CLK until CLK = '1';
        compacter;
```

```
        -- compaction d'une instruction
        elsif fetch_mc and (mc_instr /= PERVERS_BRA) then
                compacter;
        end if;

        end process COMPAC;


        end A_WATCHD;
```

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 30 mars 1992 relatif aux Etudes doctorales

VU les rapports de présentation de

. Monsieur ARLAT Jean , Chargé de Recherches C.N.R.S. - Habilité
. Monsieur LANDRAULT Christian , Directeur de Recherches C.N.R.S.

Monsieur MICHEL Thierry

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité

" **Microélectronique** "

Fait à Grenoble, le 24 février 1993

/ BV.

Pour le Président de l'INPG
et par délégation
le Directeur de l'Ecole Doctorale
J.L. LACOUME

Résumé

Cette thèse traite de la vérification en ligne, par des moyens matériels, du flot de contrôle d'un système à base de microprocesseur. Une technique de compaction est utilisée pour faciliter cette vérification (analyse de signature). La plupart des méthodes proposées jusqu'ici imposent une modification du programme d'application, afin d'introduire dans celui-ci des propriétés invariantes (la signature en chaque point de l'organigramme est indépendante des chemins préalablement parcourus). Les méthodes proposées ici, au contraire, ont comme caractéristique principale de ne pas modifier le programme vérifié et utilisent un dispositif de type processeur, disposant d'une mémoire locale, pour assurer l'invariance de la signature. Deux méthodes sont ainsi décrites. La première est facilement adaptable à différents microprocesseurs et présente une efficacité qui la place parmi les meilleures méthodes proposées jusqu'ici. La seconde méthode a été dérivée de la première dans le but de diminuer la quantité d'informations nécessaire au test. Cette dernière méthode a été implantée sur un prototype d'unité centrale d'automate programmable (avec la société Télémécanique) et son efficacité a été évaluée par des expériences d'injection de fautes. Le coût d'implantation particulièrement faible dans le cas du prototype réalisé peut permettre d'envisager une évolution de celui-ci vers un produit industriel.

Mots clefs :

Test en ligne, Vérification du flot de contrôle, Analyse de signature, Méthodes DSM, Processeurs watchdog, Processeurs à test en ligne intégré.