THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

Spécialité : **ARCHITECTURE DES SYSTÈMES INTÉGRÉS ET MICRO-ÉLECTRONIQUE**

Présentée par : **Matthieu TUNA**

Pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ PARIS VI

AUTO-TEST LOGICIEL DES SYSTÈMES INTÉGRÉS SUR PUCE (SOC)

Soutenue le : 1^{er} juin 2007

Devant le jury composé de :

M.	Christian LANDRAULT	(LIRMM)	Rapporteur
M.	Régis LEVEUGLE	(TIMA)	Rapporteur
M.	Laroussi BOUZAIDA	(ST-Microelectronics)	Examinateur
M.	Habib MEHREZ	(LIP6)	Examinateur
M.	Maurizio REBAUDENGO	(Politecnico di Torino)	Examinateur
M.	Mounir BENABDENBI	(LIP6)	Encadrant de thèse
M.	Alain GREINER	(LIP6)	Directeur de thèse

Remerciements

Je souhaite remercier en premier lieu mon directeur de thèse, Alain Greiner, directeur du département ASIM du LIP6, pour m'avoir accueilli au sein de son laboratoire. Je lui suis également reconnaissant pour sa disponibilité, ses qualités pédagogiques et scientifiques. J'ai beaucoup appris à ces côtés et je lui adresse toute ma gratitude.

J'adresse mes remerciements les plus chaleureux à Mounir Benabdenbi, maître de conférence à Paris VI, pour tous les précieux conseils qu'il m'a donnés, pour la confiance qu'il m'a témoigné et sans qui ce travail n'aurait pas vu le jour.

Je voudrais remercier les rapporteurs de cette thèse M. Christian Landrault, directeur de recherche CNRS au LIRMM, et M. Régis Leveugle, professeur à l'INPG, pour l'intérêt qu'ils ont porté à mon travail.

J'associe à ces remerciements M. Laroussi Bouzaida, responsable du département "Test Solutions" Central R&D à ST-Microelectronics Crolles, M. Habib Mehrez, professeur à l'université Paris VI, et M. Maurizio Rebaudengo, chercheur à l'université Polytechnique de Turin, pour avoir accepté d'examiner mon travail.

Je tiens à remercier tous mes camarades du laboratoire, plus particulièrement ceux qui se reconnaîtront, pour leur soutien et leur bonne humeur.

Enfin je remercie mes parents pour leur soutien au cours de ces longues années d'études et sans lesquels je n'en serai pas là aujourd'hui.

Résumé

Les finesses de gravure atteintes par les procédés de fabrication CMOS, permettent aujourd'hui d'intégrer un système complet sur une seule puce (System-on-Chip SoC). Le test de ces puces se révèle complexe. Alors que le coût de fabrication par transistor de ces puces ne cesse de décroître le coût du test reste stable. Afin de réduire le coût du test, des chercheurs ont proposé de réutiliser les capacités des SoCs à exécuter du logiciel embarqué à des fins d'auto-test.

Cette technique, l'auto-test logiciel (Software-Based Self-Test SBST), est aujourd'hui une solution alternative viable pour le test des SoCs. L'auto-test logiciel permet de se dispenser d'un équipement de test externe coûteux et possède plusieurs avantages par rapport à l'auto-test matériel (hardware Built-In Self-Test BIST). On peut distinguer dans la littérature, sur l'auto-test logiciel des SoCs, deux aspects. Le premier aspect se focalise sur l'auto-test logiciel des processeurs. Le deuxième aspect se concentre donc sur l'auto-test logiciel des autres composants du système par le processeur embarqué. Notre contribution porte sur chacun des deux aspects.

La première partie de cette thèse traite du SBST des processeurs. La littérature sur le SBST des processeurs se concentre sur les parties logiques et délaisse les composants internes de type mémoire tel que les bancs de registres et les mémoires caches. Le peu de résultats disponibles ne permet pas d'apprécier l'avantage de l'auto-test logiciel sur l'auto-test matériel de ces composants. Nous proposons le développement de programmes de test pour ces composants, programmes basés sur l'utilisation des algorithmes March. Alors que le langage d'assemblage permet un accès direct au banc de registres, les mémoires caches ne peuvent être testées que par effets de bord. Afin d'évaluer l'efficacité des programmes de test, les résultats obtenus sont comparés à ceux d'une stratégie d'auto-test matériel.

La deuxième partie de cette thèse se concentre sur le SBST des systèmes sur puce contenant des processeurs. Un SoC se construit par assemblage de coeurs préconçus tels que les processeurs, les mémoires, les DSPs, ou de blocs dédiés. Afin d'en simplifier le test, le groupe IEEE 1500 a défini un "wrapper" de test à ajouter à chaque coeur. Nous proposons l'intégration dans le système d'un micro-testeur embarqué, qui teste les coeurs munis de wrapper conforme à la norme IEEE 1500. Ainsi, nous combinons les avantages du SBST, tout en gardant la compatibilité IEEE 1500. Les performances du micro-testeur sont données, en termes de volume des données de test, de temps d'application du test et de surface additionnelle. Une comparaison avec un mécanisme d'accès au test (Test Access Mechanism TAM) traditionnel piloté par un testeur externe est aussi présentée.

Mots clés : SoC, testabilité, auto-test, SBST, IEEE 1500, wrapper, TAM, logiciel

Abstract

With the ever shrinking CMOS technologies, we are now able to build an entire system on a single chip (System-On-Chip SoC). Testing these chips appears to be complex. While the manufacturing cost of a transistor decreases continuously, the test cost remains stable. In order to lower the test cost, some researchers have proposed to reuse the SoC capability to run software for test purpose.

This technique, known as Sofware-Based Self-Test (SBST), is now a viable alternative solution for SoC testing. SBST offers many benefits, such as dispense with expensive test equipments, and brings several advantages over traditional hardware Built-In Self-Test (BIST). Two trends in SBST can be distinguished. The first one focuses on the test of processor cores. When the embedded processor test is done, it can be reused to test the remaining components in the system. Thus, the second trend concentrates on the test of processor-based systems. Our contribution is twofold.

The first part of this thesis deals with SBST of processor cores. The SBST literature focus on the processor logic parts but neglects internal memory components such as register files and memory caches. Due to the lack of available figures, the advantage of SBST over hardware BIST for these components cannot be appreciated. Therefore, we propose the development of self-test programs for these components, relying on March algorithms. While the assembly language provides direct access to the register files, memory caches can only be tested by side effects. In order to evaluate the efficiency of self-test programs, the results are compared with those of a hardware BIST strategy.

The second part of this thesis deals with SBST of processor-based systems. The SoC conception is based on the use of predesigned Intellectual Property (IP) cores. IP cores such as microprocessors, memories, DSP, or dedicated blocks are bound together to make complex SoCs. In order to simplify the test integration, the IEEE 1500 working group specified a test wrapper to be added around each IP core. Therefore, in order to take advantage of the SBST strategy on one side, and the IEEE 1500 compatibility on the other side, we propose the use of an embedded micro-tester allowing the test of wrapped cores compliant with the IEEE 1500 standard. Micro-tester performances are given, in terms of test data compaction, test application time and area overhead. A comparison with a traditional test access mechanism (TAM) driven by an external tester is also presented.

Keywords : SoC, DfT, self-test, SBST, IEEE 1500, wrapper, TAM, software.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Accroissement du nombre de transistors sur une puce

La miniaturisation des procédés de fabrication des technologies CMOS, l'augmentation du nombre de niveaux de métallisation et l'élargissement de la surface des circuits intégrés sont autant de facteurs permettant l'accroissement régulier du nombre de transistors sur une même puce. Cette augmentation croît à une vitesse exponentielle, respectant ainsi la fameuse "loi de Moore". Aujourd'hui, des puces contenant plus d'un milliard de transistors voient le jour.

Conception des systèmes sur puces (SoC)

Cette intégration considérable, permet aujourd'hui, d'intégrer un système complet sur une seule puce (System-on-Chip SoC). Afin d'appréhender la complexité de conception de tels circuits, les méthodes de conception ont rapidement migré vers la réutilisation de briques de bases préconçues (appelées coeurs ou "IP cores"). La conception d'un SoC se fait alors par assemblage de ces coeurs hétérogènes (processeurs, DSPs, mémoires, contrôleurs USB etc.) autour d'un interconnect (cross-bar, bus hiérarchisés, réseaux sur puce, etc.) permettant la communication entre les différents coeurs. Nous avons alors pu observer, dans l'industrie du semi-conducteur, une séparation entre fournisseurs de coeurs d'un côté et intégrateurs systèmes de l'autre. Afin de faciliter l'assemblage de ces coeurs, provenant de fournisseurs tiers différents, une standardisation des interfaces de communication a due être réalisée [VSIA].

Norme IEEE 1500 et test des SoCs

Une réponse de standardisation équivalente, concernant les aspects testabilités des coeurs, a été donnée par le groupe de travail IEEE 1500 [IEEE1500]. La norme IEEE 1500 prévoit l'ajout d'un "wrapper" de test autour de chaque coeur, dont l'interface a été standardisée. Les informations concernant le test du coeur (structures de test internes, vecteurs de test etc.) sont fournies dans un fichier CTL (Core Test Language [Kapur03]). Bien que la norme simplifie le test des SoCs en permettant d'en faire le test bloc par bloc, cette étape reste un facteur de coût important dans le processus de fabrication. En effet, le test des SoCs se révèle très complexe, de par la diversité des coeurs intégrés, l'inaccessibilité des noeuds internes et des jeux de test volumineux, et requiert des testeurs (Automated Test Equipment ATE) sophistiqués, et donc très coûteux.

Le test At-Speed

De plus, les nouvelles technologies introduisent des défauts affectant le circuit d'un point de vue temporel. Ces pannes temporelles ne sont pas détectables par des tests pratiqués à faible fréquence. Afin de s'assurer que le circuit fonctionne correctement à sa vitesse nominale, des tests doivent être pratiqués à cette même fréquence (test *at-speed*). Ces tests sont difficilement praticables par des testeurs de générations antérieures, sur des circuits dont les fréquences de fonctionnement sont toujours plus élevées.

L'Auto-test logiciel (SBST)

Le grand potentiel des SoCs réside dans l'intégration de coeurs programmables, supportant l'exécution de logiciel embarqué, permettant de personnaliser et d'adapter le circuit à une application donnée. Afin de réduire le coût du test, des chercheurs ont donc proposé de réutiliser les capacités des SoCs à exécuter du logiciel embarqué, à des fins d'auto-test. L'auto-test logiciel (Software-Based Self-Test SBST), de par sa caractéristique logicielle, supprime les inconvénients de l'auto-test matériel (Built-In Self-Test BIST), de par sa caractéristique embarquée, permet de pratiquer un test à la fréquence nominale du circuit (test *at-speed*) et permet de se dispenser d'un équipement de test externe onéreux.

On peut distinguer dans la littérature sur l'auto-test logiciel, deux grands courants. Le premier se focalise sur l'auto-test logiciel des processeurs. Une fois le test du processeur embarqué accompli, il peut être réutilisé pour tester les autres composants du système. Ce deuxième courant, l'auto-test logiciel des SoCs, se concentre donc sur l'auto-test logiciel des coeurs embarqués dans les SoCs.

Description du manuscrit

Notre contribution dans ce manuscrit est double. La première partie de ce manuscrit se situe dans le cadre proposé par l'auto-test logiciel des processeurs. La littérature sur le SBST des processeurs se concentre sur les parties logiques et délaisse les composants internes de type mémoire tel que les bancs de registres et les mémoires caches. Le peu de résultats disponibles ne permet pas d'apprécier l'avantage de l'auto-test logiciel sur l'auto-test matériel de ces composants. Nous proposons donc dans la première partie de ce manuscrit, une étude sur l'auto-test logiciel des bancs de registres et des mémoires caches. La deuxième partie se place dans le cadre de l'auto-test logiciel des SoCs. Nous proposons une stratégie permettant de combiner les avantages de l'auto-test logiciel tout en gardant la compatibilité des coeurs avec la norme IEEE 1500.

Le chapitre 1 introduit le test des circuits intégrés numériques. Nous étudierons dans ce chapitre quels sont les goulots d'étranglements liés à l'utilisation d'un ATE ainsi que les problèmes de conception associés à l'utilisation du BIST matériel. Ensuite, les principes de l'auto-test logiciel seront présentés, ainsi que les deux courants principaux. Nous conclurons ce chapitre par les différentes questions aux quelles nous présenterons des réponses tout au long de ce manuscrit.

Partie I : Auto-test logiciel des mémoires embarquées dans les processeurs

Le chapitre 2 introduit cette première partie du manuscrit et présente le test des mémoires.

Le chapitre 3 se focalise sur l'auto-test logiciel des bancs de registres dans les processeurs. Nous y présentons, l'état de l'art, notre approche, puis les résultats expérimentaux obtenus.

Le chapitre 4 propose une étude des caractéristiques des programmes de test des mémoires caches. Comme le précédent, ce chapitre présente tout d'abord l'état de l'art, l'approche utilisée, puis compare les résultats obtenus à ceux d'une stratégie BIST matérielle.

Finalement, le chapitre 5 conclura cette première partie, et présentera quelques points de comparaison entre l'approche SBST et BIST sur ce type de mémoires.

Partie II : Auto-test logiciel des SoCs, le micro-testeur embarqué

Le chapitre 6 introduit la deuxième partie.

Le chapitre 7 présente l'état de l'art. Les différentes stratégies d'auto-test logiciel des SoCs sont étudiées dans ce chapitre. Nous verrons que les approches utilisant des micro-testeurs embarqués permettent de rester compatible avec la norme IEEE 1500. Les points faibles et les points forts de chaque stratégie seront présentés.

Le chapitre 8 présentera notre stratégie de test basée sur l'utilisation d'un micro-testeur. Ce chapitre présente l'architecture de notre micro-testeur et expose le fonctionnement du processus de test dans ce cadre.

Le chapitre 9 traite des aspects matériels et temporels de notre approche. Ce chapitre est conclu par une présentation des fonctionnalités de test *at-speed* du micro-testeur.

Le chapitre 10 présente les performances du micro-testeur. Les plates-formes de simulation seront présentées ainsi que les résultats en terme de temps de test, de volume de données et de surface supplémentaire. Les temps de test de notre approche seront comparés à ceux d'un bus de test dédié, utilisé dans l'industrie.

Finalement le chapitre 11 conclura cette deuxième partie.

Une conclusion générale récapitulera les points importants présentés dans ce document et exposera les perspectives à ces travaux.

Chapitre 1

Problématique

La fabrication des circuits intégrés est une suite d'étapes très complexes. C'est pourquoi le bon fonctionnement de chaque circuit après fabrication n'est pas garanti. Le test des circuits intégrés est donc un passage obligé, permettant d'effectuer le tri entre circuits correctement fabriqués et circuits défectueux.

Le test peut être effectué à l'aide d'un testeur externe (ATE), ou, du matériel dédié au test peut être ajouté sur la puce elle-même, afin de rendre le circuit auto-testable. On parle alors d'auto-test matériel ou de test intégré. Malheureusement ces deux techniques ont leurs limites. En effet, la complexité des circuits actuels, tels que les systèmes sur puce (SoC), poussent les testeurs dans leurs limites. Nous présenterons dans ce chapitre quels sont les goulots d'étranglements liés à l'utilisation de ces équipements. Bien que le test intégré permette de réduire l'utilisation d'un testeur externe, nous verrons que cette technique ne peut pas être appliquée à tous les types de circuits.

D'un autre côté, le grand potentiel des SoCs réside dans l'exécution de logiciel embarqué, permettant de modeler leur utilisation. Pourquoi ne pas réutiliser cette capacité à exécuter du logiciel embarqué à des fins d'auto-test ? Nous présenterons l'auto-test logiciel (SBST), une alternative au test externe et au test intégré. Les deux grands courants que sont, l'auto-test logiciel des microprocesseurs et l'auto-test logiciel des SoCs seront présentés de façon succincte.

Nous présenterons pour conclure, les questions auxquelles ce manuscrit tente de répondre, sur l'auto-test logiciel des microprocesseurs dans un premier temps, sur l'auto-test logiciel des SoCs dans un second temps.

1.1 Le test des circuits intégrés

Défauts de fabrication

Les différentes étapes de fabrication des circuits intégrés sont très complexes. Pendant leur fabrication, des défauts dus aux procédés de fabrication, aux matériaux, ou introduits lors de l'encapsulation dans le boîtier peuvent survenir. Ces défauts sont observés par la manifestation d'une erreur. Les erreurs peuvent se produire en présence de certains stimuli.

Modèles de fautes

Afin de pouvoir créer des stimuli de façon automatique, des abstractions des défauts sont réalisées sous la forme de modèles de fautes. Plus le modèle de faute s'abstrait de la technologie employée plus la migration technologique est aisée. Un modèle de faute populaire est celui du collage permanent d'une ligne à la valeur 0 ou 1 (Stuck-At fault). Lors de la génération des vecteurs de test, d'autres modèles peuvent être pris en compte comme le modèle de couplage (Coupling fault) pour le test des mémoires, ou les fautes temporelles (Delay fault) pour la logique. Ce dernier modèle est de plus en plus utilisé. En effet, la réduction des largeurs des transistors et des lignes d'interconnexions, le fonctionnement des circuits à des fréquences élevées rendent les circuits de plus en plus sensibles aux défauts temporels.

Test des circuits intégrés

Le test intervient après la fabrication du circuit afin de vérifier si son comportement est conforme au comportement attendu. Des vecteurs de test (binaires) sont appliqués aux entrées du circuit sous test. Les réponses du circuit sont comparées aux réponses attendues. Le circuit est considéré bon si les réponses correspondent. La figure 1.1 présente les principes de base du test numérique.

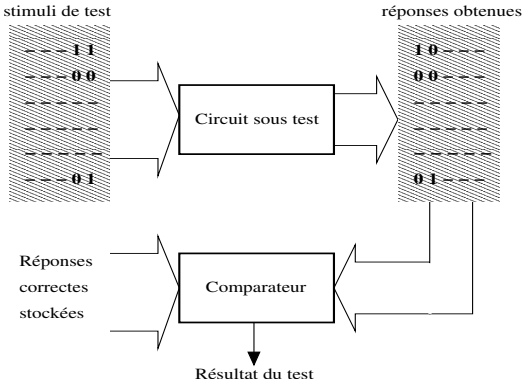


FIG. 1.1: Principes de base du test numérique [Bushnell00]

Afin d'appliquer les vecteurs de test et de comparer les réponses obtenues à celles désirées, des équipements de test sont utilisés : les testeurs. Le test peut être réalisé par un testeur externe, ou,

du matériel peut être intégré sur la puce afin d'effectuer ces opérations de test. On parle d'auto-test matériel ou de test intégré.

Test externe

Les équipements de test externe, plus connus sous le nom de ATE (Automatic Test Equipment), permettent d'appliquer les vecteurs de test sur les entrées du circuit sous test (CST ou DUT/CUT pour Device/Circuit Under Test), d'analyser les réponses obtenues sur les sorties du DUT et de marquer le DUT comme correct ou fautif. Cependant l'évolution des circuits intégrés rend rapidement les performances des ATEs obsolètes, bien que toujours plus onéreux.

Test intégré (BIST)

L'idée du BIST (Built-In Self-Test) est de concevoir un circuit qui peut se tester lui-même et ainsi déterminer s'il a été fabriqué sans défauts ou non. Ce type de solution doit être pensé dès la conception du circuit. En effet, le BIST consiste en l'introduction de logique supplémentaire consacrée uniquement à l'auto-test de la puce. Ce matériel supplémentaire doit être capable de générer les vecteurs de test, et de fournir un mécanisme permettant de déterminer si les réponses aux stimuli fournis par le circuit sous test correspondent à un circuit sans défaut. La figure 1.2 présente l'architecture de base d'un circuit contenant un BIST.

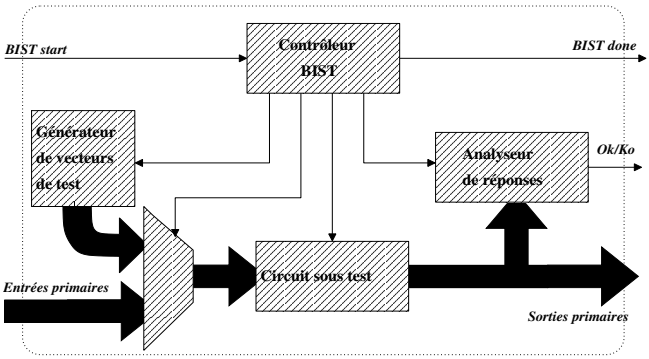


FIG. 1.2: Architecture du BIST

On peut distinguer les différents éléments supplémentaires suivants :

- un générateur de vecteurs de test, généralement un LFSR (Linear Feedback Shift Register),
- un analyseur de réponses, contenant un compacteur de réponses tel qu'un MISR (Multiple Inputs Shift Register) et un comparateur,
- de la logique de contrôle (bloc contrôleur BIST),
- de la logique permettant l'isolation des entrées (le multiplexeur).

Un avantage majeur de l'auto-test est de pouvoir effectuer un test à la vitesse nominale du circuit (*test at-speed*) permettant ainsi la detection des fautes temporelles.

1.2 Les SoCs : des coeurs au système

Grâce a une intégration toujours plus importante du nombre de transistors sur une même puce, les concepteurs peuvent désormais intégrer un système complet sur une seule puce (System-on-Chip "SoC"). La conception d'un tel circuit se fait par assemblage de différents blocs, appelés coeurs, que sont les processeurs, les DSPs, les RAMs, ROMs, contrôleurs USB, PCI, Ethernet, etc. En tant que système, ces coeurs communiquent entre eux via un interconnect système qui peut être un bus, un cross-bar, un réseau (NoC), etc. La figure 1.3 présente un exemple de SoC : la plate-forme Nexperia utilisé par NXP [NXP].

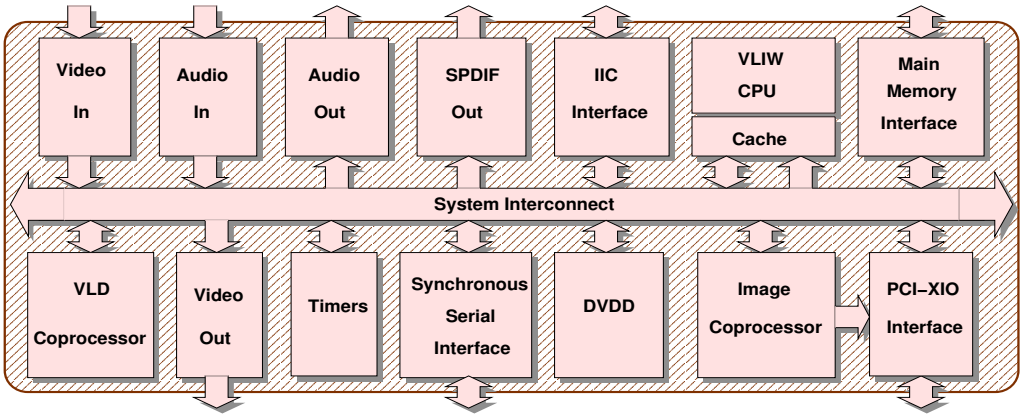


FIG. 1.3: Exemple d'architecture d'un SoC : la plate-forme Nexperia PNX 1300 [NXP]

Ces coeurs sont généralement disponibles sous forme d'une description synthétisable écrite en VHDL ou Verilog, ou sous la forme d'un "layout". La flexibilité d'utilisation du coeur dépend donc de la forme sous laquelle il est fourni. Trois catégories peuvent être dégagées :

- *Soft cores :* disponible sous la forme d'une description RTL (Register Transfert Level) synthétisable. Ceci implique que l'utilisateur du coeur est responsable de son implémentation matérielle.
- *Hard cores :* de l'autre côté du spectre, ces coeurs sont disponibles sous forme de layout (i.e. tel que le format GDSII). Ils sont optimisés en performance, en consommation, et en taille, et sont donc livrés pour une technologie donnée. Ce type de coeur permet de ne fournir que très peu de détails sur l'implémentation du coeur protégeant ainsi les droits de propriété intellectuelle (intellectual property "IP"). C'est pourquoi ces coeurs sont souvent référencés sous le terme "IP cores".
- *Firm cores :* est une catégorie intermédiaire, ces coeurs sont disponibles sous forme d'une combinaison de code RTL synthétisable, d'une netlist de portes complète ou partielle, des références de la bibliothèque cible ainsi que des détails du "floorplan".

La conception d'un SoC souligne l'importance de la réutilisation de briques de base, matérielles comme logicielles, afin d'augmenter la productivité. Ayant d'un coté les fournisseurs de coeurs, de l'autre les intégrateurs système, un effort de standardisation à été réalisé [VSIA] afin d'intégrer facilement ces blocs hétérogènes venant de fournisseurs différents sur la même plate-forme.

1.3 Test des SoCs

Les systèmes sur puces intègrent tous les composants sur le même morceau de silicium. Ainsi, tous les composants seront donc fondus et testés en même temps. Ce n'est donc pas aux fournisseurs d'effectuer le test du composant mais aux intégrateurs système. Néanmoins, le fournisseur doit-il livrer un coeur testable (chaînes de scan insérées, points de test, et/ou BIST, vecteurs de test etc.), ou est-ce à l'intégrateur système d'insérer ces structures de test ?

Nous verrons dans cette section comment les responsabilités concernant le test et la testabilité sont partagées entre fournisseurs et intégrateurs, ainsi que les conséquences au niveau architectural.

Le test d'un SoC est généralement une alliance entre test externe et test intégré. Nous présenterons les points bloquants liés à l'utilisation de testeurs externes et les problèmes introduits par le test intégré.

1.3.1 Architecture pour le test des SoCs

Comme nous l'avons précisé, un coeur est fourni sous forme de code RTL synthétisable, de netlist ou bien sous forme d'un layout. Pour les catégories *soft cores* et *firm cores*, le test peut être introduit par l'intégrateur système. En revanche, dans le cas des *hard cores* le fournisseur doit avoir au préalable inséré les structures de test dans le coeur puisqu'il ne peut être modifié par l'intégrateur. Il faut donc convenir d'une responsabilité partagée, concernant la testabilité, entre le fournisseur de composants et l'intégrateur système. Des standards ont été réalisés au niveau de la conception du système [VSIA]. Une réponse de standardisation équivalente au niveau de la testabilité des coeurs a été apportée, permettant ainsi de structurer le test des SoCs. L'IEEE TTTC (Test Technology Technical Council) a donc formé le groupe de travail IEEE 1500 [IEEE1500].

Au niveau coeur

L'initiative IEEE 1500 propose l'élaboration d'un anneau d'isolation ou "wrapper" de test autour de chaque coeur. Le wrapper de test doit fournir les principaux modes suivants :

- *Mode fonctionnel :* le wrapper est transparent, le coeur est utilisé de façon fonctionnelle dans le système.
- *Mode test interne :* le wrapper isole logiquement le coeur du voisinage, les stimuli sont appliqués aux entrées, les réponses sont capturées sur les sorties.
- *Mode test externe :* le wrapper permet de tester les interconnexions entre deux coeurs.

Le comité IEEE 1500 a standardisé l'interface du wrapper ainsi que l'accès aux fonctionnalités test du coeur. La figure 1.4 présente l'architecture d'un wrapper IEEE 1500.

Il existe deux niveaux de conformité à la norme IEEE 1500. Le coeur est fourni avec son wrapper de test compatible avec la norme. On dit alors que le coeur est "IEEE 1500 Compliant". Ou alors, le wrapper peut être conçu par l'intégrateur système. Dans ce cas le coeur doit être "IEEE 1500 Ready". Afin de véhiculer les informations propres au test du coeur, les auteurs de la norme ont créé le langage CTL (Core Test Language) [Kapur03]. Le fichier CTL, fourni avec le coeur, décrit toutes

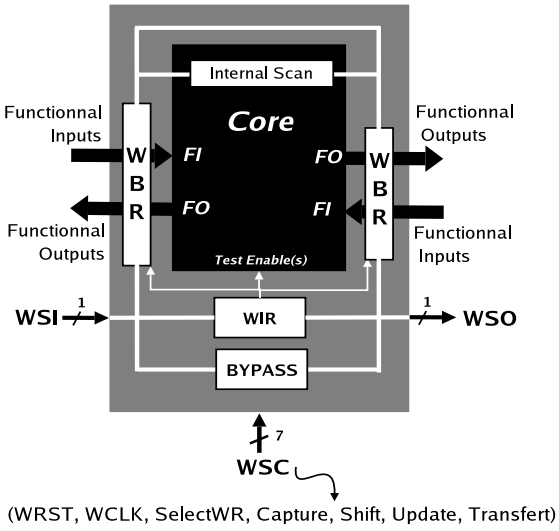


FIG. 1.4: Architecture d'un wrapper IEEE 1500

les spécificités propres au test du coeur pour que soit, il s'intègre directement dans le système lorsque celui-ci est fourni avec le wrapper, soit l'intégrateur puisse générer le wrapper et l'intégrer.

Cependant, alors que la norme standardise l'interface de test des coeurs, elle laisse libre l'intégrateur système de concevoir le mécanisme permettant le contrôle de ces wrappers ainsi que les mécanismes d'acheminement des vecteurs de test vers ces wrappers.

Au niveau système

Une fois les coeurs équipés de leur wrapper de test, l'intégrateur système doit prévoir le matériel permettant de stocker les vecteurs de test ainsi que les structures permettant d'acheminer ces vecteurs jusqu'aux wrappers. La figure 1.5 présente une vue conceptuelle de l'architecture pour le test des SoCs. Voici une définition des différentes ressources nécessaires au test d'un système sur puce :

- *Test Source :* matériel fournissant les vecteurs de test. La source peut être interne, un LFSR, un compteur ou une ROM contenant les vecteurs de test. Elle peut être externe, une mémoire off-chip ou traditionnellement l'ATE.
- *Test Sink :* matériel analysant les vecteurs réponses. Le sink peut être interne, comme un analyseur de signature ou un processeur embarqué. Elle peut être externe, comme l'ATE.
- *Test Access Mechanism (TAM) :* structure définie par l'utilisateur permettant de transporter les stimuli de la source au coeur, et les réponses, du coeur au sink. Cette structure n'est pas standardisée et est laissée a l'appréciation de l'intégrateur système.
- *Test Controller :* coordonne l'exécution, l'ordonnancement du test du système. En général, le contrôleur de test est partagé entre les contrôleurs BIST internes et l'ATE externe.

Chaque coeur peut être testable de façon individuelle par test externe ou par BIST. Ainsi, le test d'un SoC est réalisé en combinant des sources internes et externes, et de la même façon, l'analyse des

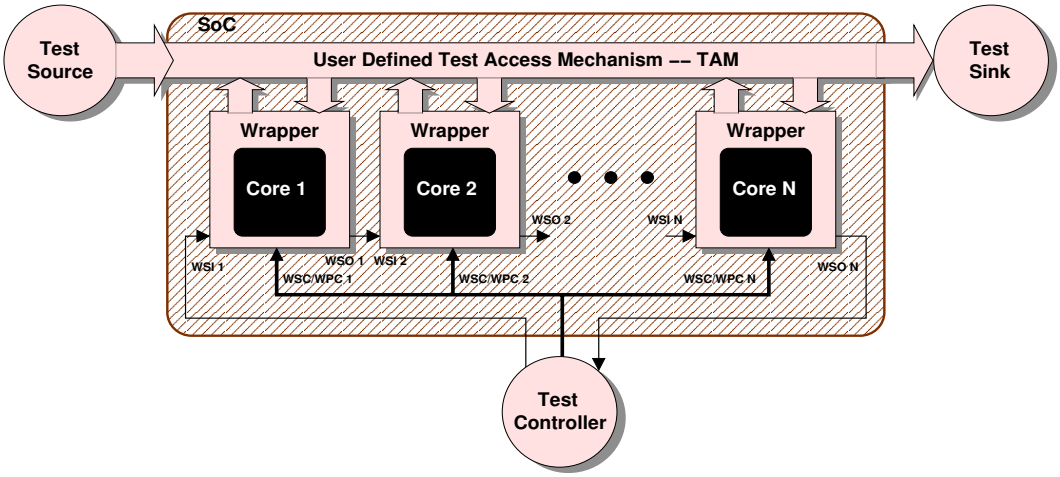


FIG. 1.5: Architecture de test pour les SoCs

vecteurs se fait par une conjugaison de sinks intégrés et dispersés dans la puce, et de l'ATE.

Nous avons vu que la norme IEEE 1500 permettait la réutilisation du test au niveau coeur, mais qu'elle laissait libre l'intégrateur système dans le choix du mécanisme permettant d'acheminer les vecteurs de test (TAM) aux coeurs wrappés. C'est pourquoi, on peut trouver dans la littérature pléthore de TAMs. Néanmoins, deux types d'architectures différentes peuvent être distinguées. La première solution consiste à ajouter un bus spécifique, dédié au test. La deuxième, réutilise au maximum les ressources fonctionnelles du système comme TAM afin de réduire le matériel dédié au test.

- **Ressources dédiées pour le TAM.** Le premier type d'approche crée un bus de test ad-hoc afin de transporter les vecteurs de test [Varma98, Marinissen98, Benabdenbi00]. Ce type de TAM a l'avantage d'être flexible, paramétrable, permettant une optimisation en fonction des contraintes que sont le temps de test, l'accessibilité au niveau wrapper, le nombre d'entrées/-sorties de la puce dédiées au test, le nombre de broches du testeur, la puissance dissipée etc. Malheureusement, la largeur du bus (i.e. la bande passante) dépend du nombre de plots dédiés au test qui peuvent être ajoutés sur la périphérie de la puce, ainsi que du nombre de broches testeur disponibles. Ainsi, la bande passante du bus (ressource partagée entre tous les coeurs testés) est en général assez faible, augmentant le temps de test.
- **Ressources fonctionnelles utilisées comme TAM.** Afin de limiter l'ajout de plots de test et réduire le matériel dédié au test, une seconde approche consiste à réutiliser les ressources fonctionnelles du système. Le plus souvent l'interconnect système est réutilisé [Feige98, Cota03, Nahvi04, Amory06]. Ce type d'approche permet de minimiser la surface additionnelle due au TAM par réutilisation de l'interconnect système. Cependant, le problème de ces approches réside toujours dans l'utilisation de wrapper de test spécifiques (car branchés sur l'interconnect), et donc, non standards. Ce type de TAM, du fait de l'utilisation de wrappers de test spécifiques, ronge l'effort de standardisation et d'interopérabilité du test au niveau coeur. De plus, ces approches font l'hypothèse importante que chaque coeur est connecté à l'interconnect système.

1.3.2 Problèmes liés à l'utilisation d'un testeur externe

Lorsqu'un SoC de plusieurs millions de transistors est testé par un ATE, différents facteurs doivent être pris en considération afin de limiter le coût du test. Les différents goulots d'étranglement responsables du coût du testeur sont :

- **L'accessibilité.** La difficulté de tester s'intensifie avec l'accroissement de la densité des transistors. Ceci est du au fait que les noeuds internes de la puce sont de plus en plus difficile d'accès. Cette évolution est inéluctable. En effet, le nombre de plots d'entrées/sorties est proportionnel au périmètre de la puce, alors que le nombre de transistors est proportionnel à l'aire de la puce. Il n'y a aucun moyen d'inverser cette tendance, signifiant que tester une puce depuis l'extérieur deviendra de plus en plus impraticable. Alors que le nombre d'entrées/sorties tend à augmenter régulièrement, le nombre de broches des testeurs reste limité.
- **Le volume des jeux de test.** Logiquement cette intégration de plus en plus forte a un impact sur le volume des données de test. La taille des jeux de test se répercute directement sur le coût du test. D'un côté, le temps de test dépend directement du volume à appliquer, d'un autre côté, le volume des données se répercute sur la profondeur mémoire du testeur. Or, le coût de ces équipements dépend du nombre de canaux de test et de la profondeur mémoire associée à chacun d'eux.
- **La précision des testeurs.** Pat Gelsinger (Intel) commente, dans l'article [Gelsinger00], l'évolution de la précision des testeurs par rapport aux vitesses de fonctionnement des circuits. Les testeurs externes du début des années 80 avaient une résolution excédentaire par rapport aux exigences du composant testé. A cette époque une perte de rendement faible était due à la marge de tolérance du testeur. En une vingtaine d'années, alors que la vitesse des DUTs a été multipliée par cent, la précision des testeurs n'a été accrue que d'un facteur dix. La projection de cette tendance montre que d'ici quelques années une perte de rendement inacceptable sera due seulement à l'imprécision des testeurs.
- **Le test at-speed.** Les nouvelles technologies CMOS introduisent de plus en plus de défauts à caractère résistif résultant dans des pannes temporelles [Bennetts06]. De plus, les circuits fonctionnent avec plusieurs domaines d'horloge opérant à des fréquences de plus en plus élevées [Vermeulen01]. C'est pourquoi, afin d'obtenir un test de qualité, un test pratiqué à la fréquence nominale du circuit (test at-speed) est aujourd'hui requis. Or une horloge rapide augmente le coût du testeur. Les testeurs bas coût ne fournissent généralement pas de telles horloges. De plus, certains SoCs communiquent avec leur environnement à une fréquence très faible. Les plots sont lents comparés à la fréquence de fonctionnement interne. Il serait donc impossible à un testeur de transmettre une horloge rapide à de tels circuits [Beck05].

1.3.3 Problèmes liés à l'utilisation du test intégré

Afin de réduire les coûts liés à l'utilisation d'un testeur externe, une solution est d'embarquer dans le SoC des coeurs munis de BIST. Ainsi ces coeurs ne nécessitent pas de testeurs précis, réduisent le volume des jeux de test, augmentent l'accessibilité aux noeuds internes et permettent un test at-speed. Malheureusement le BIST est une technique engendrant d'autres problèmes :

- **La conception.** Un des inconvénients les plus significatifs du BIST se situe lors de la phase de conception, en particulier pour l'introduction du LBIST (Logic BIST). En effet, les sources de génération de valeurs indéterminées (Xs) doivent être circonscrites pendant la phase de test afin de ne pas corrompre la signature calculée, les conflits sur le bus doivent être gérés, les points de test doivent de préférence ne pas être insérés sur les chemins critiques etc. Ainsi, en plus de la conception et de la vérification des opérations propres au système, il faut faire face à la conception et la vérification des opérations propres au système BIST. Deux systèmes doivent être conçus, avec le système BIST au-dessus du système voulu [Stroud02].
- **Le taux de couverture.** La longueur du test ainsi que le taux de couverture sont déterminés par simulation de faute. Cependant, la simulation de faute est une tâche très chronophage puisque la complexité est en $O(N^2)$ voire en $O(N^3)$, ou N est le nombre de portes dans le circuit (alors que la complexité de la simulation logique est en $O(N)$) [Goel80]. Ainsi, à chaque changement dans le bloc de génération de vecteur, une simulation de faute doit être relancée pour vérifier si le taux de couverture attendu est atteint. De plus les techniques BIST ne garantissent pas un taux de couverture aussi complet que peut l'être celui obtenu lorsque les vecteurs de test sont générés par ATPG [Hetherington99, Chen00b, Chen01b].
- **La surface supplémentaire.** La circuiterie, largement intrusive, ajoutée par le BIST impacte négativement les performances du circuit testé. Premièrement, cette circuiterie supplémentaire de test a un coût en surface. Des surfaces plus larges ont pour conséquence de diminuer le nombre de puces sur le wafer, et d'augmenter la surface susceptible de contenir des défauts de fabrication. Deuxièmement, l'augmentation de l'aire influe négativement sur les performances du circuit.

Ces contraintes ont fait que le BIST s'est surtout développé pour des structures régulières telles que les mémoires. En effet, de par la structure connue et l'utilisation d'algorithmes réguliers (tel que les algorithmes March), l'insertion du BIST a pu être complètement automatisé et les tests sont de très bonne qualité.

BIST pour petites mémoires embarquées

Bien que le BIST pour les mémoires (MBIST) apparaît comme une solution efficace, le problème de la surface supplémentaire subsiste. La surface supplémentaire engendrée par les contrôleurs BIST pour mémoires n'est pas proportionnelle à la taille des mémoires testées. Ainsi, l'ajout d'un contrôleur BIST dédié à chaque petite mémoire (de l'ordre de quelques Ko) engendre une telle surface supplémentaire que ce type de solution n'est absolument pas envisageable.

La figure 1.6 illustre la surface supplémentaire dédiée à la circuiterie BIST. La première courbe (carré noir) est pour un BIST dédié à une mémoire. Une surface non négligeable peut être sauvée lorsque le compteur d'adresses de la mémoire peut être réutilisé (courbe à diamants blancs). Afin de réduire la surface engendrée par le contrôleur BIST, il peut être partagé par deux mémoires identiques (courbe à carrés blancs). Nous pouvons constater qu'un contrôleur BIST pour une mémoire de 1Kbit représente quasiment 60% de la surface totale du circuit.

C'est pourquoi, en général, ces petites mémoires sont wrappées et partagent un même contrôleur BIST, permettant ainsi la réduction de la surface engendrée par ce type de technique. La figure 1.7

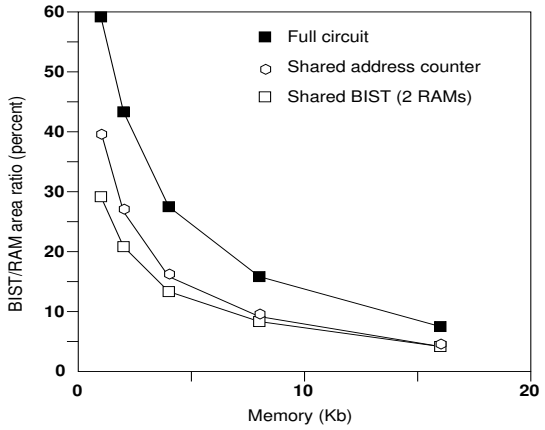


FIG. 1.6: Surface supplémentaire engendrée par un BIST [Nadeau-Dostie90]

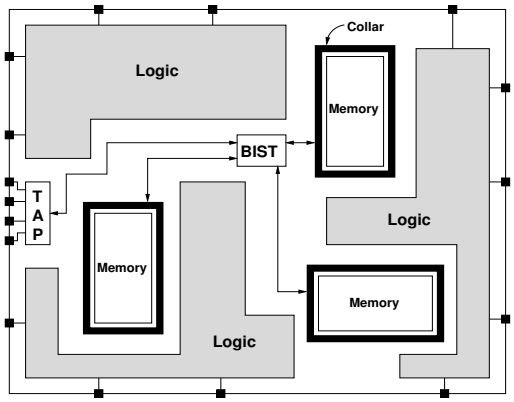


FIG. 1.7: Contrôleur BIST partagé entre plusieurs mémoires [Nadeau-Dostie00].

expose ce concept où un unique contrôleur BIST est utilisé pour tester trois mémoires différentes. Le contrôleur BIST peut envoyer les bits de données et d'adresses en parallèle ou en série.

- Approche parallèle : envoyer tous les bits en parallèle permet d'effectuer un test de même qualité qu'un BIST dédié. Néanmoins un désavantage, aussi lié à la surface supplémentaire, est le routage de la filasse nécessaire pour amener la nappe de fils de données, d'adresses et de contrôles jusqu'à chaque mémoire. En effet, ces mémoires sont larges (utilisent des mots de 32 bits) et éparpillées à travers le système. La surface supplémentaire engendrée par le routage d'une telle filasse est non négligeable.

- Approche série : le contrôleur peut envoyer les bits en série permettant ainsi la réduction de la surface liée à la filasse. Malheureusement, les temps de test sont alors très longs, et le test n'est plus pratiqué at-speed.

Les processeurs embarqués dans les SoCs contiennent des petites mémoires telles que les bancs de registres et les mémoires caches. Or, l'introduction de matériel BIST pour tester ces petites mémoires impacte négativement les performances des processeurs, qui sont généralement optimisés en fréquence, en consommation, en surface etc.

1.3.4 Conclusion sur le test des SoCs par test externe et test intégré

Le test des SoCs

Le test d'un système sur puce pose aujourd'hui beaucoup de défis. Bien que de nouvelles structures aient été mises au point pour le test de ces systèmes (mécanisme d'accès au test TAM, wrapper de test, norme IEEE1500), le test de chaque coeur se fait toujours par de vielles recettes que sont le test externe et le test intégré.

Par testeur externe

Les différents goulots d'étranglement, présentés dans la section 1.1, montrent qu'une approche de type test externe se dirige vers une impasse. En effet, l'inaccessibilité des noeuds internes depuis l'extérieur de la puce suit une croissance inéluctable. Les jeux de test hypertrophiés conduisent à l'utilisation de mémoires de plus en plus larges, et donc, à un coût toujours plus élevé des ATEs. Un autre problème de taille est celui de la précision des testeurs. Les ATEs ne supportent plus la cadence imposée par les circuits intégrés introduisant des pertes de rendement bientôt inacceptables. Alors que le test at-speed est aujourd'hui une nécessité, il est difficilement praticable depuis l'extérieur de la puce.

Par auto-test matériel

L'auto-test matériel (BIST), bien que réduisant considérablement les coûts du test, est surtout utilisé pour le test de structures régulières comme les mémoires (MBIST), mais reste peu utilisé pour les circuits logiques (LBIST) (voir section 1.1). En effet, les coeurs intégrant du BIST souffrent du matériel ajouté. Un circuit contenant un BIST a exigé un effort de conception supplémentaire non négligeable. La surface supplémentaire, augmente la surface active sensible aux défauts et dégrade les performances du circuit. Enfin, le taux de couverture, qui est souvent inférieur à celui obtenu par ATPG, est rapporté au prix de très longues séances de simulation de fautes. Bien que souvent employé pour le test des mémoires, nous avons pu constater que la surface engendrée pour le test de petites mémoire était rédhibitoire, surtout lorsque ces petites mémoires sont embarquées dans des circuits orientés performance tel que les processeurs.

L'auto-test logiciel

Néanmoins, si l'on regarde d'un point de vue fonctionnel l'architecture d'un SoC, leur potentiel réside dans la vaste gamme de fonctionnalités embarquées disponibles et surtout, dans l'exécution de logiciel embarqué permettant de modeler à loisir leur type d'utilisation. Pourquoi ne pas faire exécuter au système, un programme, dédié au test de la puce ? Ainsi, ce programme spécifique pourrait réutiliser les ressources disponibles sur la puce afin de s'auto-tester. **L'auto-test logiciel** minimiserait l'utilisation d'équipements externe ainsi que l'ajout de matériel dédié au test.

1.4 Auto-test logiciel

Nous avons vu que le test des circuits intégrés se complexifie à chaque nouvelle génération et amalgame les techniques de test intégrés et de test externes. Cependant, les techniques reposant sur les équipements de test externe (Automatic Test Equipment ATE) se dirigent vers une impasse, et le rôle de ces testeurs externes est en passe de changer. Face aux problèmes posés par le test depuis l'extérieur de la puce, une solution est d'intégrer davantage de fonctionnalités consacrées au test, fournissant ainsi des puces de plus en plus auto-testables. Malheureusement, les techniques d'auto-test matériel introduisent de la circuiterie supplémentaire dédiée au test, et ces circuits souffrent de ce matériel annexé.

Avec l'avènement des systèmes sur puce (SoC), leurs capacités à exécuter du logiciel embarqué a éveillé la curiosité de certains chercheurs de la communauté test. En effet, peut-on envisager le détournement des fonctionnalités embarquées dans le système, par exécution d'un programme spécifique, à des fins d'auto-test ?

Depuis quelques années, des chercheurs se sont donc penchés sur la question, créant ainsi une nouvelle alternative au test des circuits intégrés, connu sous le nom de "**Software-Based Self-Test**" (**SBST**) et que nous appellerons **auto-test logiciel**. La capacité de calcul des processeurs embarqués et les nombreuses fonctionnalités disponibles dans le système, donnent la possibilité de transférer à l'intérieur de la puce quasiment toutes les fonctionnalité du test sans adjonction de matériel supplémentaire. Le processeur est la pièce centrale de l'auto-test logiciel car il est réutilisé comme infrastructure de base pour le test de fabrication. Après son auto-test, le processeur peut être utilisé pour tester les autres composants du système.

Les avantages de l'auto-test logiciel sont les suivants (voir [Gizopoulos04] pour plus de détails) :

- Le SBST est une méthode de test *non intrusive*. Elle ne nécessite pas (ou peu) de modifications en vue de la testabilité du circuit.

- Le SBST est une méthode *bas coût*. Nul besoin d'équipement de test externe performant et donc de coût prohibitif. En effet, des testeurs bas coût, fonctionnant à de faibles vitesses et ayant peu de canaux de test peuvent être utilisés.

- Le SBST effectue un test *at-speed*. Puisque le système est utilisé pour s'auto-tester, la fréquence nominale est utilisée afin de fournir un test *at-speed*, fournissant ainsi une très bonne qualité de test. Ce type de test est en général la caractéristique des techniques d'auto-test.

- Le SBST est très *flexible* et *programmable*. Le fait d'être une méthode logicielle permet par de simples ajouts ou modifications de code d'augmenter les capacités du test (comme augmenter le taux de couverture, diminuer le temps de test etc.).

L'auto-test logiciel peut être divisé en deux grandes parties. La première et plus ancienne, se focalise sur l'étude du test des processeurs. La deuxième plus récente, propose des techniques génériques pour le test des autres coeurs embarqués dans le SoCs.

1.4.1 Auto-test logiciel des microprocesseurs

L'auto-test logiciel des microprocesseurs n'est pas nouveau. Ce problème est adressé depuis le milieu des années 1970. Il s'applique indifféremment sur les processeurs individuels (type Pentium), comme pour les coeurs de processeurs embarqués dans les SoCs (type MIPS). Dans les deux cas, le test s'opère de la même façon.

La figure 1.8 présente les concepts de base de l'auto-test des processeurs.

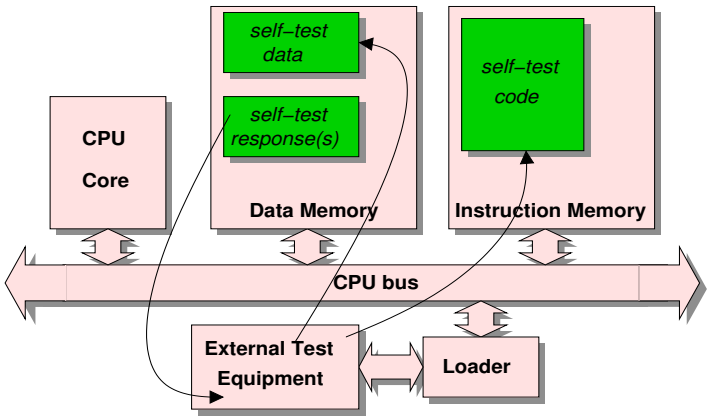


FIG. 1.8: Concepts de l'auto-test logiciel des processeurs d'après [Gizopoulos04]

L'application d'une stratégie d'auto-test logiciel d'un coeur de processeur pour le test de fabrication consiste en les étapes suivantes :

1. Les routines d'auto-test (self-test code) sont chargées dans la mémoire instruction embarquée via un équipement de test externe. Le testeur externe doit avoir accès au bus interne ou directement aux mémoires. Pour ce faire, un chargeur (loader sur la figure 1.8) doit être introduit dans la puce pour permettre la communication entre le testeur externe et le système. [Bernardi04] prévoit l'utilisation d'un IP d'infrastructure (Infrastructure IP "I-IP" [Zorian02]) connecté à l'interconnect système d'un côté, et permettant la communication avec l'ATE de l'autre. Un chargeur (cache-loader) est utilisé dans [Parvathala02] afin de charger directement les mémoires caches du processeur testé. Une alternative peut être de stocker ces routines de façon permanente dans une ROM ou une mémoire flash afin d'opérer un test en ligne (on-line testing).
2. Les données (self-test data) sont chargées de la même manière. Elles contiennent des paramètres, les variables et les constantes du code embarqué, les patterns de test, et les réponses attendues qui seront comparées aux réponses obtenues.
3. Lorsque toutes ces données sont chargées (ou non dans le cas d'un test en ligne), le processeur peut commencer à exécuter le programme. Ainsi, les patterns de test sont appliqués aux composants internes du processeur via les instructions afin d'en détecter les fautes. Les réponses obtenues sont enregistrées dans les registres et/ou dans la mémoire de données. Les réponses peuvent être compactées.
4. Finalement, après l'exécution du programme de test, le testeur externe, via le loader, récupère les réponses pour être évaluées.

1.4.2 Auto-test logiciel du SoC

Une fois le test du processeur accompli, il peut être utilisé pour tester les autres coeurs du système. L'auto-test logiciel d'un SoC par le processeur peut être vu comme une extension de l'auto-test logiciel du processeur seul.

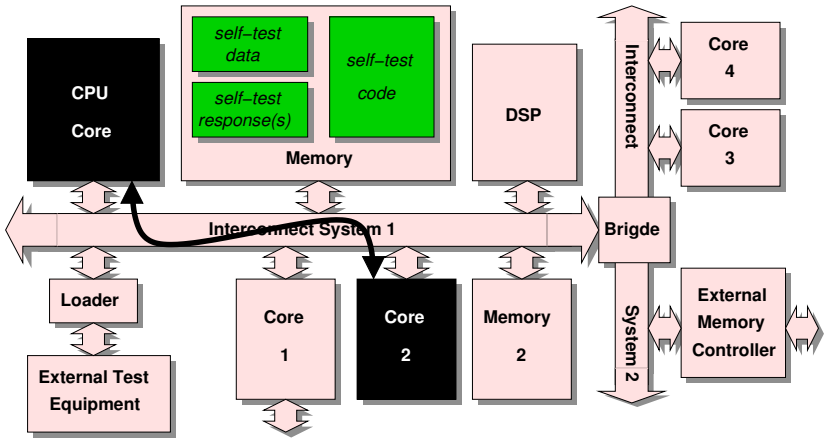


FIG. 1.9: Concepts de l'auto-test logiciel des SoCs par le processeur embarqué

De la même façon que pour l'auto-test du processeur, les routines d'auto-test ainsi que les données sont chargées dans les mémoires embarquées depuis le testeur externe. Ensuite, ces routines sont exécutées at-speed, le processeur est utilisé comme générateur de vecteur de test, appliquant ces vecteurs à travers l'interconnect système aux différents coeurs à tester. Les réponses sont ensuite récupérées par le processeur et rangées en mémoire. Le processeur a la possibilité de compacter les réponses. Finalement, le testeur externe récupère la/les réponse(s) pour évaluer si la puce contient des défauts ou non.

La littérature traitant de l'auto-test logiciel des SoCs est plus maigre que celle des processeurs. En effet, les SoCs sont plus récents que les microprocesseurs d'une part, et d'autre part, les plates-formes très diverses, compliquent la recherche de méthodes génériques de test. On peut distinguer deux approches dans la littérature. La première vise un composant particulier. Les approches testant les microprocesseurs embarqués en sont un exemple, mais on peut citer le test des mémoires, des bus systèmes, des DSPs etc. Le deuxième type d'approche cherche une stratégie générique pour le test des composants du système sans regard de leur spécificité. En général, ce type d'approche réutilise les structures DfT présentes dans le coeur comme les chaînes de scan, les wrappers de test etc.

1.5 Conclusion

Le test des SoCs est principalement conduit par un testeur externe. Malheureusement, afin de gérer l'augmentation du nombre d'entrées/sorties, des fréquences de fonctionnement plus élevées, et des jeux de test toujours plus volumineux, des machines de test toujours plus onéreuses sont nécessaires. Afin de réduire les inconvénients dus au test externe, des coeurs peuvent être munis de BIST, mais ces

techniques souffrent des pénalités qu'engendre l'introduction de matériel supplémentaire et ne sont utilisées que pour quelques types précis de circuits.

Une alternative a été présentée : l'auto-test logiciel (SBST). De par son caractère embarqué, il réduit les problèmes liés au test externe. De par son caractère logiciel, il minimise les problèmes liés à l'introduction de matériel dédié au test. Le SBST peut être divisé en deux parties distinctes : le test des microprocesseurs et le test des SoCs. Les contributions de ce manuscrit portent sur ces deux domaines.

Partie 1 : Auto-test logiciel des microprocesseurs

Les processeurs sont composés de blocs hétérogènes et, plus particulièrement, embarquent des petites mémoires comme les bancs de registres et les mémoires caches. Or nous avons vu que le BIST pour mémoire, bien qu'étant une approche intéressante, engendrait pour des petites mémoires une surface bien souvent rédhibitoire.

Sachant que le SBST pour les processeurs propose une solution "0 surface additionnelle", peut-on envisager d'effectuer des routines de test spécifiques à ces petites mémoires embarquées ? Comparé à une approche BIST classique, le SBST de ces petites mémoires propose-t-il la même qualité de test ? En terme de temps de test, le SBST est-il plus, ou moins, intéressant qu'une approche BIST classique ?

La première partie de ce manuscrit répond à ces questions en proposant des techniques d'auto-test logiciel spécifiques aux bancs de registres et aux mémoires caches, et compare les résultats obtenus à ceux d'une approche matérielle BIST classique.

Partie 2 : Auto-test logiciel des SoCs

D'un côté la norme IEEE 1500 assure une intégration aisée des coeurs dans le flot de conception d'un SoC, permet une automatisation des tâches réduisant ainsi les coûts du test et favorise la conception en vue du test (Design for Test DfT) améliorant ainsi la qualité du test. D'un autre côté l'auto-test logiciel pour les SoCs réduit la surface nécessaire dédiée au test par réutilisation des ressources fonctionnelles embarquées, réduit l'utilisation d'équipement de test externe, et permet un test de qualité (auto-test).

Comment combiner l'approche SBST et la norme IEEE 1500 ? En d'autre termes, comment effectuer un test de coeurs muni d'un wrapper IEEE 1500 de façon logicielle et embarquée ?

Une réponse possible est d'intégrer dans le système un composant dédié au test de ces coeurs muni de wrappers : le micro-testeur. Nous présenterons dans la deuxième partie de ce manuscrit une stratégie reposant sur l'utilisation d'un micro-testeur embarqué. Nous essayerons de répondre aux questions suivantes : Comment le micro-testeur embarqué teste les coeurs couronnés de wrapper IEEE 1500 ? Quelle est la surface engendrée par un tel composant ? Bien que réutilisant les structures scan du coeur, est-il toujours possible d'effectuer un test at-speed de ces composants ? Finalement les micro-testeurs rivalisent-ils, en terme de temps de test, avec les approches de test classiques guidées par ATE ?

Première partie

Auto-test logiciel des microprocesseurs : ses mémoires embarquées

Chapitre 2

Introduction

Les circuits récents regorgent de petites mémoires (de l'ordre de quelques kilo-octets) embarquées dans le système. Les microprocesseurs, de plus en plus nombreux sur les plates-formes, contiennent des bancs de registres et des mémoires caches. Les mémoires de type FIFO sont énormément utilisées pour la communication entre coeurs. Ces petites mémoires, généralement larges (orientées mot), sont physiquement éparpillées dans le système.

Comme nous l'avons montré dans le chapitre précédent, les solutions BIST pour mémoires, bien qu'efficaces, ajoutent une surface beaucoup trop élevée pour les mémoires de petites tailles. Un BIST pour un banc de registres (mémoire de 1 kilo-bits) occupe 60% de la surface totale. Les mémoires caches (de l'ordre de quelques kilo-octets), sont composées de plusieurs bancs mémoires. Afin de réduire la surface, un contrôleur BIST partagé peut être utilisé. Malheureusement ce type de solution, comme nous l'avons vu, amène des désavantages comme des temps de test très longs (approche série), un supplément de surface dû au routage de la filasse (approche parallèle), etc.

L'auto-test logiciel des microprocesseurs, présenté précédemment, supprime les inconvénients liés à l'utilisation du BIST. C'est pourquoi, dans cette première partie, nous allons nous intéresser à l'auto-test logiciel des bancs de registres et des mémoires caches. Nous allons proposer le développement de routines spécifiques à ces deux types de mémoires. Le chapitre 3 se focalisera sur l'étude des bancs de registres, le chapitre suivant portera sur le test des mémoires caches. Le dernier chapitre concluera sur l'interêt d'une approche SBST par rapport aux approches BIST pour ces mémoires.

Avant d'aller plus loin, nous allons tout d'abord présenter succinctement l'architecture des mémoires ainsi que leurs tests. Nous présenterons les modèles de fautes associés et les algorithmes utilisés.

2.1 Architecture des mémoires SRAM

Les générateurs de mémoires ou les bibliothèques de composants fournissent une large variété de mémoires SRAM telles que les FIFOs, les ROMs, les mémoires caches et les bancs de registres. Ces mémoires peuvent être divisées en trois grandes parties fonctionnelles (voir figure 2.1) :

- La matrice de cellules mémoires
- Le décodeur d'adresses
- La logique de lecture/écriture

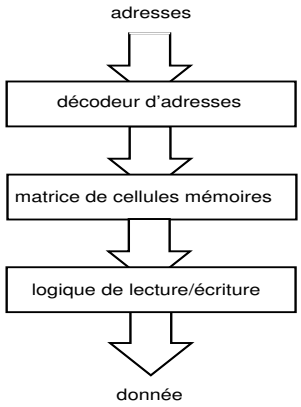


FIG. 2.1: Modèle fonctionnel simplifié d'une mémoire

La matrice de cellules mémoires est constituée de cellules mémoires SRAM à 6 transistors comme le montre la figure 2.2. Les transistors T1-T2 forment un inverseur, couplé en opposé avec le second inverseur T3-T4. Les transistors de passage T5 et T6 permettent les opérations de lecture et d'écriture. Le décodeur ligne sélectionne une ligne de cellules en activant son signal ligne de mot (LM). Ce signal est connecté aux transistors de passage de toutes les cellules de cette ligne. Pour sélectionner une cellule particulière dans cette ligne, le décodeur colonne active les signaux de ligne de bit (LB et $\overline{LB}$) correspondants.

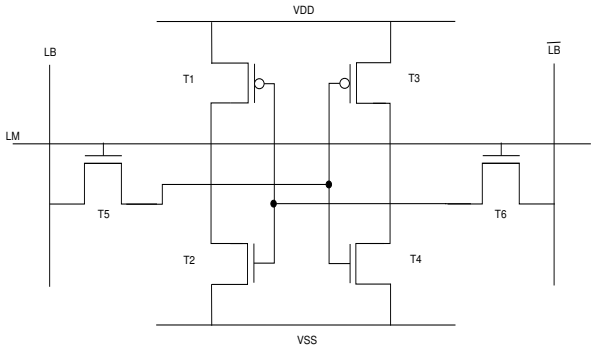


FIG. 2.2: Une cellule mémoire SRAM à 6 transistors

2.2 Test des mémoires SRAM

Des tests appropriés à ces circuits permettent de minimiser les coûts du test tout en garantissant une détection élevée des défauts physiques. Les algorithmes de test pour mémoires sont basés sur l'utilisation de modèles de fautes propres à ces structures.

Nous verrons dans cette section les modèles de fautes qui sont utilisés dans la suite du manuscrit, ainsi que les tests utilisés.

2.2.1 Modèles de fautes

La littérature foisonne de travaux concernant la création des modèles de fautes de mémoires. Nous décrivons ici, brièvement, les modèles de fautes utilisés :

Les fautes dans la matrice mémoire

- Faute de collage (Stuck-At Fault SAF) : la valeur logique de la cellule est toujours 0 (SA0) ou toujours 1 (SA1).
- Faute de transition (Transition Fault TF) : une cellule ne peut effectuer la transition de la valeur 0 à la valeur 1 ou de la valeur 1 à la valeur 0 lors de l'écriture.
- Faute de couplage (Coupling Fault CF) : une faute de couplage implique deux cellules (couplante et couplée). On distingue les CFs dépendants de l'état de la cellule couplante, et ceux dépendants de la transition de celle-ci :
    - Faute de couplage d'état (State Coupling Fault CF$_{st}$) : la cellule couplante C_i est dans un état donné E_x forçant la cellule couplée C_j dans un état E_y.
    - Faute de couplage idempotente (Idempotant Coupling Fault CF$_{id}$) : une transition positive ou négative de la cellule couplante C_i force le contenu de la cellule couplée C_j dans état E_y.
    - Faute de couplage inverse (Inverting Coupling Fault CF$_{inv}$) : une transition positive ou négative de la cellule couplante C_i inverse l'état de la cellule couplée C_j indépendamment du contenu de celle-ci.

Les fautes dans le décodeur d'adresses (Address Fault AF)

Une faute dans le décodeur d'adresses représente une erreur dans le décodage de l'adresse passée en entrée de la mémoire. Selon [Goor91], les fautes fonctionnelles du décodeur d'adresses peuvent être classées parmi les types de comportements suivants :

- En présence d'une certaine adresse, aucune cellule ne peut être accédée.
- En présence d'une certaine adresse, plusieurs cellules sont accédées simultanément.
- Une certaine cellule ne peut jamais être accédée.
- Une certaine cellule peut être accédée avec plus d'une adresse.

Les fautes dans la logique de lecture/écriture

Les fautes de la logique de lecture/écriture peuvent être transposées en fautes dans la matrice mémoire (faute de collage, faute de transition et faute de couplage) [Goor91], et peuvent donc être détectées par les tests dédiés à celle-ci.

2.2.2 Les tests March

Des tests appropriés à des modèles de fautes pertinents assurent la détection des défauts physiques conduisant à un test de qualité. Afin de détecter ces fautes propres aux RAMs, des tests spécifiques tels que les tests March ont été développés. Un test March consiste en une séquence finie d'éléments March. Un élément March est une séquence finie d'opérations (ou primitives March) devant être appliquées à une cellule mémoire avant de passer à la cellule suivante. Ainsi, $\Uparrow\ (r0, w1)$ est un élément March. $r0$ et $w1$ sont des primitives March. La séquence d'adresses d'un élément March peut s'effectuer dans un ordre croissant ($\Uparrow$), décroissant ($\Downarrow$), ou ($\Updownarrow$) si l'ordre des adresses est indifférent. Une primitive peut être une écriture d'un 0 ou d'un 1 d'une cellule ($w0$ ou $w1$), ou une lecture d'un 0 ou d'un 1 d'une cellule ($r0$ ou $r1$).

Le tableau 2.1 liste quelques algorithmes que nous utiliserons par la suite. Le tableau 2.2 présente les fautes détectées ainsi que la complexité de ces algorithmes.

Algorithme	Séquence de l'algorithme
MATS	$\{ \Updownarrow (w0); \Updownarrow (r0, w1); \Updownarrow (r1) \}$
MATS+	$\{ \Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0) \}$
MATS++	$\{ \Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0) \}$
MARCH X	$\{ \Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Updownarrow (r0) \}$
MARCH C-	$\{ \Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0) \}$

TAB. 2.1: Différents algorithmes March.

Algorithme	Fautes détectées						Complexité
	SAF	AF	TF	CF$_{in}$	CF$_{id}$	CF$_{st}$	
MATS	Toutes	Quelques					4.n
MATS+	Toutes	Toutes					5.n
MATS++	Toutes	Toutes	Toutes				6.n
MARCH X	Toutes	Toutes	Toutes	Toutes			6.n
MARCH C-	Toutes	Toutes	Toutes	Toutes	Toutes	Toutes	10.n

TAB. 2.2: Modèles de fautes détectés par différents algorithmes March.

Test des mémoires orientées mots

Les bancs de registres ainsi que les mémoires caches sont accessibles par mots de N bits (N > 1) et non par bit. Pour ces mémoires orientées mots les opérations de lecture et d'écriture impliquent la lecture et l'écriture d'un mot de N bits appelé *mot de fond* (Data Background DB). Ainsi, l'algorithme MATS orienté mot devient $\{\Updownarrow(\mathrm{w}a)\,;\Updownarrow(\mathrm{r}a,\,\mathrm{w}\bar{a})\,;\Updownarrow(\mathrm{r}\bar{a})\}$ où a est le mot de fond et $\bar{a}$ le mot de fond inverse. Les fautes n'impliquant qu'une cellule (par exemple SAF) restent détectables par n'importe quel couple de mots de fond. Cependant, les fautes impliquant deux cellules (par exemple CF) introduisent le problème de détection de fautes entre deux cellules d'une même adresse (i.e. du même mot). Dans [Goor98], Van De Goor divise les fautes entre cellules mémoires en fautes inter-mots et en fautes intra-mot. L'unique façon de détecter ce dernier type de faute est d'exécuter le test March plusieurs fois en utilisant des mots de fond différents à chaque itération. Dans [Dekker90] une méthode permettant la construction des mots de fond pour la détection des fautes de couplage (CFs) est donnée. Le tableau 2.3 présente les six mots de fond nécessaires à la détection des fautes de couplage intra-mots pour une mémoire ayant des mots de 32 bits.

numéro	Mot de fond	
	normal	inverse
1	0x00000000	0xFFFFFFFF
2	0x0000FFFF	0xFFFF0000
3	0x00FF00FF	0xFF00FF00
4	0x0F0F0F0F	0xF0F0F0F0
5	0x33333333	0xCCCCCCCC
6	0x55555555	0xAAAAAAAA

TAB. 2.3: Différents mots de fond pour la détection des fautes de couplage intra-mot.

Chapitre 3

Auto Test Logiciel du Banc de Registres

Dans ce chapitre nous allons nous intéresser à l'auto-test logiciel du banc de registres.

La première partie de ce chapitre sera consacrée à une présentation de la structure des bancs de registres, ainsi qu'à une étude de l'état de l'art. Nous verrons quelles sont les différentes approches utilisées dans la littérature afin d'effectuer un auto-test logiciel des bancs de registres de microprocesseurs.

La deuxième partie présentera notre approche. Nous présenterons une façon systématique de générer des programmes de test testant le banc de registres. Le langage machine MIPS sera utilisé pour les exemples. Nous verrons quels sont les pièges à éviter (i.e. processeur pipeline, partitionnement du banc de registres) et comment gérer ces différents problèmes.

Finalement nous présenterons la plate-forme nous permettant de valider notre approche. Nous exposerons les résultats obtenus, en terme de taux de couverture, de temps de test et de tailles de programmes. Les résultats obtenus seront comparés à l'état de l'art.

Un dernière section récapitulera nos travaux sur l'auto-test logiciel du banc de registres.

3.1 État de l'art

3.1.1 Contexte

On peut dégager de la littérature récente sur l'auto-test logiciel des microprocesseurs, deux approches duales. La première approche, orientée instructions, manipule des séquences d'instructions afin d'obtenir la couverture de fautes maximale sur les composants du processeur. L'approche duale, orientée composants, part des valeurs à appliquer aux composants permettant le taux de couverture maximal, et génère la séquence d'instructions permettant d'appliquer ces valeurs aux composants internes.

Approche orientée instructions

Dans ce type d'approche, seul le jeu d'instructions suffit au développement du programme de test. Aucune connaissance de la structure interne du processeur n'est requise.

Jacob Abraham et al. dans [Thatte80] et [Shen98], utilisent un graphe représentant le microprocesseur au niveau RTL (Register Transfer Level) pour la génération de programmes de test. Pour un modèle de microcontrôleur 8 bits de type Intel 8085, 360 000 instructions ont été nécessaires pour obtenir 86,7% de taux de couverture. Une génération automatique et aléatoire du programme de test est proposée dans [Batcher99] et [Parvathala02]. Dans [Batcher99], le flux d'instructions est généré par un composant matériel dédié sur la puce. Dans [Parvathala02], le programme de test est généré par logiciel. Pour la première approche, 220 000 instructions ont été nécessaires pour obtenir un taux de couverture de 95% sur un processeur DLX. La deuxième teste un processeur complexe (Pentium 4 et Itanium) et n'obtient que 70% de taux de couverture. Dans [Corno01b, Corno01a], une approche évolutionnaire pour la génération du programme est proposée. Le programme généré, d'environ 600 instructions, permet d'obtenir 85% de taux de couverture sur le microcontroleur 8 bits Intel 8051.

Approche orientée composants

La deuxième catégorie suggère une approche structurelle. Basées sur la maxime "diviser pour régner", les techniques développées ciblent les composants du processeur, et permettent un réglage fin dans le développement du test bas niveau (portes logiques).

Li Chen et Sujit Dey proposent dans la suite d'articles [Chen99, Chen00a, Chen00b, Chen01b] le test du processeur 8 bits PARWAN [Navabi93]. Les auteurs s'attaquent au test de 3 composants (boite à opération, décaleur et compteur de programme). Le programme de test réalisé pèse un peu plus d'1ko pour un taux de couverture de 91%. Kranitis et al., dans [Kranitis02b, Kranitis02a], proposent de développer manuellement des routines de test spécifiques à chaque composant du processeur. Les programmes développés obtiennent 91% de taux de couverture avec seulement 400 instructions sur le PARWAN. Dans [Kranitis03b, Kranitis03c] le programme teste un processeur de type RISC : le Plasma/MIPS pipeline à 3 étages. Avec moins de 1000 instructions et seulement 6000 cycles, le taux de couverture obtenu est d'environ 95%.

Les approches orientées instructions ont l'avantage d'un développement de test bas coût. Le niveau élevé d'abstraction permet une intervention humaine très limité. Cependant, les faibles taux de couverture dus à leur philosophie pseudo-aléatoire sont compensés par l'augmentation de la taille de la séquence de test. D'un autre côté, les approches orientées composants demandent, souvent, la disponibilité de la netlist et d'experts compétents ayant une connaissance approfondie du processeur sous test. Ces techniques obtiennent en contrepartie un taux de couverture élevé pour un programme de test relativement léger.

3.1.2 L'auto-test logiciel des bancs de registres

Intéressons nous maintenant à un composant particulier du processeur : le banc de registres.

Les processeurs modernes utilisent un grand nombre de registres d'utilisation générale. Ces registres sont regroupés en un composant appelé banc de registres. De facto, le banc de registres est un des plus gros composant du processeur, et représente une part importante de la surface totale occupée sur silicium. C'est pourquoi un test de qualité sur ce type de composant est important.

Avant de présenter l'état de l'art, faisons un point sur l'architecture de ce composant.

Architecture des bancs de registres

La plupart des bancs de registres des microprocesseurs RISC pipeline possède trois ports. Un pour l'écriture, deux pour la lecture. Si l'on considère une architecture 32 bits, le cas général est un banc de registres de 32 registres de 32 bits. Ainsi, le banc de registres contient 1024 éléments mémoires (1 kilo-bits/128 octets) accessibles en écriture et en lecture par mot de 32 bits. La figure 3.1 illustre l'interface d'un banc de registres à 3 ports. Afin d'en réduire la taille et la consommation, le banc de registres est conçu comme une mémoire orientée mot multi-ports [Rabaey03].

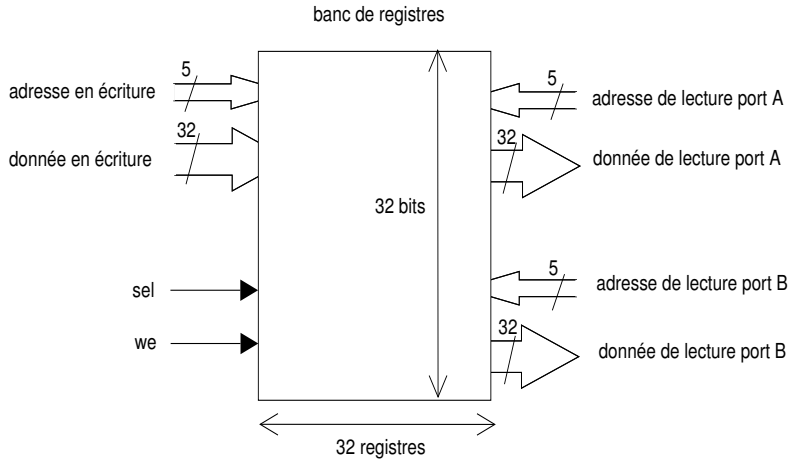


FIG. 3.1: Interface d'un banc de registres à 3 ports (1 port d'écriture et 2 ports de lecture A et B)

Le banc de registres vu comme une netlist de portes logiques

L'auto-test logiciel du banc de registres des microprocesseurs RISC est effectué dans la suite d'articles [Kranitis03b, Kranitis03c, Paschalis04, Hatzimihail05]. Cependant, notre attention se portera plus particulièrement sur l'article [Kranitis03a] qui présente de façon détaillée l'implémentation du test.

Kranitis utilise la procédure décrite dans [Gizopoulos04], qui consiste à créer des routines de test spécifiques pour chaque sous-composant du processeur. La structure du banc de registres étudié dans cet article est présentée dans la figure 3.2.

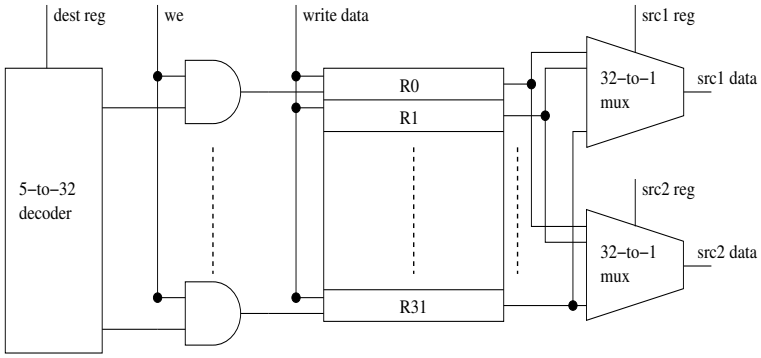


FIG. 3.2: Structure interne d'un banc de registres proposé dans [Kranitis03a]

Trois sous-composants peuvent être dégagés : les multiplexeurs (n-vers-1) de sorties, les registres de stockage et le décodeur d'adresses du registre destination.

- Test des multiplexeurs de sorties. Puisque les multiplexeurs n-vers-1 de sorties sont eux-mêmes constitués de multiplexeurs 2-vers-1, ils peuvent être testés de façon exhaustive en terme de fautes de collage. En effet la séquence suivante présente les différentes combinaisons de valeurs sur les entrées (S, A, B) permettant le test exhaustif d'un multiplexeur 2-vers-1 : (0,0,1), (0,1,0), (1,0,1) et (1,1,0) où S est le signal de sélection du multiplexeur et A, B sont les deux données d'entrées qui sont sélectionnées quand S=0 et S=1 respectivement. En général, pour un mux n-vers-1, le jeu de test déterministe est linéaire, $2n$ vecteurs de test sont requis pour obtenir un taux de couverture de fautes de collage de 100%.
- Test des registres. Les registres utilisés sont décomposés en un ensemble de bascules. Ainsi, seulement deux vecteurs de test sont nécessaires. Puisque ces registres sont complètement observables lorsque les multiplexeurs de sorties propagent leur valeurs, le taux de couverture de fautes de collage est lui aussi de 100%.
- Test du décodeur d'adresses. Ce composant n'est pas testé dans l'article, étant considéré comme peu prioritaire de par sa petite surface occupée sur silicium comparé aux deux sous-composants précédents.

Dans l'article [Paschalis04], les auteurs nous renseignent un peu plus quant à l'implémentation du programme. Ils nous indiquent principalement que le test du banc de registres est effectué en deux phases. En effet, le programme de test à besoin d'utiliser quelques registres pour sa propre exécution.

Ces registres ne peuvent donc pas subir le test. Ainsi, durant la première phase de test, une première moitié du banc de registres est testée, l'autre est utilisée pour l'exécution du programme. Durant la seconde phase, l'inverse est effectué.

Les résultats rapportés dans [Hatzimihail05, Kranitis05, Paschalis04, Kranitis03a] montrent que pour obtenir un taux de couverture d'environ 99% de fautes de collage, des programmes de test de 1Ko à 3 Ko pour une exécution de l'ordre du millier de cycles sont nécessaires. Evidemment ces résultats dépendent du processeur cible.

Les problèmes liés à cette approche sont :
– Les bancs de registres embarqués dans les microprocesseurs sont en général conçus comme des mémoires. Ainsi, par exemple, la partie mémorisante du banc de registres est une matrice de cellules mémoires à 6 transistors. Utiliser la méthode présentée sur de tels bancs de registres, et donc ne considérer que le modèle de faute de collage, n'amènent à considérer que 50% des fautes se produisant dans ces composants [Dekker90].
– Ce type d'approche requiert un développement très bas niveau, et donc une connaissance précise de l'architecture interne du banc de registres.

Le banc de registres vu comme une mémoire

Van de Goor et Verhallen [Goor92] se sont attaqués aux tests du banc de registres et des mémoires caches du processeur RISC Intel i860^{TM}. Le test du processeur est effectué à partir des techniques décrites dans [Thatte80] et [Brahme84]. Afin de couvrir le test des fonctionnalités des mémoires embarquées, Van de Goor et Verhallen étendent le modèle fonctionnel présenté dans [Thatte80, Brahme84]. Ils proposent des modèles de fautes propres à la classe des mémoires, ainsi que les tests correspondants.

Nous nous intéresserons ici au test du banc de registres, le test des mémoires caches étant traité dans la section suivante.

Comme nous l'avons précisé, Van de Goor et Verhallen prennent en compte la structure mémoire des bancs de registres. Ainsi, les modèles de fautes étudiés sur ce composant sont :

1. Les fautes de collage à 0 et à 1.
2. Les fautes de transition.
3. Les fautes de couplage.
4. Les fautes du décodeur d'adresses.

Les auteurs proposent alors d'utiliser les tests spécifiques aux mémoires (comme les tests March [Goor91]) afin de détecter les fautes décrites ci-dessus. Le banc de registres étant une mémoire orientée mot, il est précisé que les tests employés doivent être répétés pour une suite particulière de mots de fond [Goor98].

Les problèmes liés à cette approche sont :
– Aucun renseignement n'est donné sur l'implémentation du test.
– Comme nous l'avons vu dans l'approche précédente [Kranitis03a, Paschalis04], le test doit être exécuté au moins en deux temps. Or, nous verrons dans la suite de ce manuscrit qu'une telle division

de ce type de test n'est pas sans impact sur la détection des fautes impliquant plusieurs cellules comme les fautes de couplage ou certaines fautes d'adresses. Pourtant, ce problème n'est pas traité.
– Finalement, aucune valeur numérique sur la taille des programmes ni sur leur temps d'exécution n'est dévoilée.

Plus récemment, Zhou et Wunderlich proposent dans l'article [Zhou06], l'utilisation de l'algorithme MATS pour le test du banc de registres. Une implémentation de cet algorithme est présentée dans l'article. Il est précisé que des tests March plus complexes pourraient être utilisés.

Les problèmes liés à cette approche sont :
– Seul le test MATS est présenté, et donc, seules les fautes de collage sont visées.
– Bien que le test MATS détecte théoriquement 100% des fautes de collage, il est reporté dans l'article que le taux de couverture atteint sur ce modèle de faute n'est que de 98%.
– Aucune approche systématique n'est présentée.

3.1.3 Conclusion

L'approche proposée dans [Kranitis03a, Paschalis04, Gizopoulos04], bien que présentant une méthode complète et chiffrée, ne saurait convenir à la plupart des bancs de registres conçus comme des mémoires.

D'un autre côté, Zhou et Wunderlich dans [Zhou06], Van de Goor et Verhallen dans [Goor92] proposent l'utilisation des tests March. Malheureusement, aucune approche systématique n'est présentée et la taille des programmes de test n'est pas connue. De plus, [Paschalis04] précise qu'un partitionnement du banc de registres est nécessaire afin d'utiliser des variables ou des valeurs stockées dans ce composant. Or, bien que le partitionnement réduise la détection de certaines fautes (CFs et AFs) comme nous le verrons par la suite, [Goor92] n'en fait pas état.

Nous proposons donc, à partir de la section suivante, une étude de l'implémentation des algorithmes March sur le banc de registres. Une approche systématique est proposée, afin de générer automatiquement des programmes de test à partir de n'importe quels algorithmes March. Les chiffres donnés dans [Gizopoulos04] nous servirons de points de comparaison, puisque les méthodes employées dans ce livre sont connues pour leur efficacité en terme de temps de test et de taille de programme.

3.2 Implémentation des algorithmes March

Nous proposons dans cette section une approche systématique permettant de générer de façon automatique des programmes de test implémentant des algorithmes March pour le test du banc de registres.

Un algorithme March se construit par assemblage d'éléments March. Un élément March, comme nous l'avons vu précédemment, est constitué de primitives March (primitives d'écritures et de lectures) ainsi que d'un séquencement des adresses.

Nous proposons une étude systématique, dans le sens où nous verrons comment implémenter chaque primitive March en une ou plusieurs intructions assembleur. Nous verrons comment la répétition des ces intructions nous permet d'implémenter le séquencement des adresses, et donc de créer un élément March. Le programme implémentant un certain algorithme March ne se fera que par simple aboutement des différentes séquences de programme implémentant chacune un élément March.

Nous verrons donc tout d'abord, comment implémenter les primitives March, suivi du séquencement des adresses. Finalement nous verrons comment résoudre les problèmes "producteurs/consommateurs" introduits par les machines pipelines.

Avant d'aller plus loin, nous devons présenter le langage machine MIPS. En effet, les programmes de test du banc de registres doivent manipuler chaque registre de façon précise. C'est pourquoi le choix du langage s'est porté sur un langage d'assemblage. Nous considérerons une architecture de type RISC : l'architecture MIPS. L'implémentation d'un algorithme March en langage machine MIPS passe par le choix d'instructions permettant d'implémenter les primitives d'écriture et de lecture sur le banc de registres.

3.2.1 Le langage machine MIPS

Toutes les instructions du MIPS ont une taille de 32 bits. Le banc de registres contient 32 registres nommé `$0` pour le registre 0 à `$31` pour le registre 31. Le jeu d'instructions (Instruction Set Architecture ISA) des processeurs de type MIPS est divisé en 3 formats différents : R, I et J (voir figure 3.3).

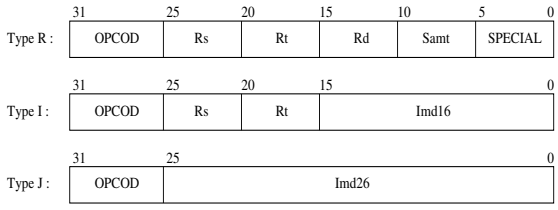


FIG. 3.3: Les 3 formats du jeu d'instructions MIPS

Le format R est utilisé pour les opérations arithmétiques et logiques utilisant deux registres source (Rs/Rt) et un registre destination (Rd). Le format I est utilisé pour les opérations d'accès mémoire (load/store), les opérations arithmétiques et logiques utilisant un immédiat, ainsi que les sauts conditionnels courts (branch if). Le format J est quant à lui utilisé pour les sauts inconditionnels longs (jump).

3.2.2 Les primitives March

Primitive d'écriture

L'opération "wa" signifie écrire la valeur a (mot de fond) dans le mot visé de la mémoire. Dans le cas du banc de registres, cette opération consiste à charger dans le registre visé la valeur a. Les formats I et R fournissent trois moyens différents d'effectuer cette opération. Le mot de fond a peut provenir de trois sources différentes : il peut être encodé dans les instructions du programme, chargé depuis la mémoire ou chargé depuis un autre registre.

- **Depuis les instructions.** Le format I permet de ranger directement un opérande immédiat de 16 bits dans une instruction. Comme le mot de fond considéré a une longueur de 32 bits, il doit être découpé en deux tronçons de 16 bits. Ainsi, afin d'écrire un mot de fond dans un registre, au moins deux instructions de format I sont nécessaires. Le listing 3.1 ci-dessous présente l'écriture d'un mot de 32 bits dans le registre 5.

Listing 3.1: Implémentation de la primitive d'écriture

```
1 lui  $5,  Immd1          // charge Immd1 dans les 16 bits de poids fort
2 ori  $5,  $5,   Immd2    // $5 <- (Immd1<<16) | Immd2
```

`Immd1` et `Immd2` correspondent respectivement aux 16 bits de poids forts et aux 16 bits de poids faibles du mot de fond.
Coût de la méthode : nécessite 2 cycles pour être appliquée et 2 mots dans la mémoire des instructions.

- **Depuis la mémoire.** Une unique instruction permet de charger une donnée depuis la mémoire : `lw Rd, Immd(Rs)`. Le mot rangé dans la mémoire à l'adresse `Rs+Immd` est écrit dans le registre pointé par le registre `Rd`. Utiliser cette instruction implique deux choses. Premièrement, le mot de fond a du être rangé en mémoire auparavant. Deuxièmement, l'adresse du mot de fond doit être présente dans le registre pointé par Rs. Ce dernier point est important puisqu'il implique le fait que ce registre, contenant l'adresse, ne peut à la fois détenir ce type d'information et subir le test en même temps. C'est pourquoi nous placerons ce registre dans une partie du banc de registres que nous appelons *zone réservée*. Cette *zone réservée* du banc de registres ne subira pas le test.
Coût de la méthode : le coût en cycles dépend du nombre de cycles nécessaires pour accéder à la mémoire de données. En terme de place mémoire occupée, l'opération nécessite 1 instruction dans la mémoire des instructions et l'ensemble des mots de fond utilisés en mémoire de données.
- **Depuis un registre.** La dernière méthode charge le mot de fond depuis un autre registre référence. L'instruction `or Reg, Ref, Ref` copie le contenu du registre `Ref` dans le registre `Reg`. Le registre `Ref` doit être initialisé par une des deux méthodes précédemment décrites et doit être placé dans la *zone réservée*.
Coût de la méthode : cette primitive d'écriture ne consomme que 1 mot en mémoire des instructions et s'exécute en 1 cycle.

Primitive de lecture

Une opération de lecture "ra" implique deux étapes distinctes : une lecture et une comparaison du mot lu avec le mot de fond attendu (a). Plusieurs techniques peuvent être adoptées afin de réaliser ces opérations de lecture et comparaison :

- **Stocker les réponses en mémoire.** Tous les mots lus du banc de registres peuvent être stockés en mémoire. L'instruction "store word" `sw Rd, Immd(Rs)` permet d'effectuer cette opération. La comparaison peut être effectuée par le testeur externe qui viendra lire les valeurs en mémoire et les comparera à celles attendues. Comme pour l'instruction `lw` vue précédemment, un registre contenant l'adresse mémoire (`Rs`) où sera stockée la donnée, est nécessaire. Même cause même conséquence, ce registre ne peut subir le test et doit être placé dans la *zone réservée*.
  Coût de la méthode : cette méthode est très coûteuse en mémoire. Après chaque opération de lecture la mémoire contient une "image" du banc de registres. D'un autre côté, le choix de cette méthode s'avère très utile pour effectuer un diagnostic.
- **Calcul d'une signature.** Cette méthode est similaire aux techniques matérielles de type MISR. Pour effectuer cette opération il est nécessaire d'initialiser un registre dans la *zone réservée* et d'utiliser une séquence d'instructions qui va calculer une signature. Afin d'éviter l'aliasing, cette séquence d'instructions doit être choisie avec attention puisque tous les registres lus contiennent la même valeur. La dernière étape consiste en la comparaison de la signature calculée avec celle attendue.
  Coût de la méthode : un unique mot (la signature calculée) est stocké en mémoire. Cependant, la séquence d'instructions calculant la signature peut s'avérer gourmande en temps (en nombre de cycles).
- **Lecture et comparaison immédiate.** Dans le jeu d'instruction MIPS, une instruction permet de comparer deux registres. Ainsi, un registre contient le mot de fond et est utilisé comme référence, l'autre registre est le registre testé. Le registre référence est rangé dans la *zone réservée* du banc de registres. L'instruction *Branch If not Equal* s'utilise comme ceci : `bne Reg, Ref, label`. Le registre `Ref` est le registre référence, `Reg` est le registre testé. Si `Ref` est différent de `Reg` le programme se branche à la partie `label`. Le sous programme pointé par `label` gère les cas où la comparaison a échouée.
  Coût de la méthode : l'avantage de cette méthode est qu'une seule instruction est nécessaire pour effectuer la lecture ainsi que la comparaison. De plus, le programme s'interrompt à la première erreur rencontrée.

3.2.3 Séquencement des adresses

Chaque élément March s'exécute suivant un parcours croissant des adresses ($\Uparrow$) ou décroissant ($\Downarrow$). Dans le cas d'un banc de registres, l'adresse correspond au numéro du registre destination. Le listing 3.2 présente un exemple d'implémentation de l'élément March $\Uparrow$(r0, w1). Les registres `$30` and `$31` sont placés dans la *zone réservée* et contiennent respectivement un mot de fond et le mot de fond inverse correspondant.

Listing 3.2: Séquencement des instructions

```
1            ...
2   bne  $5,  $30,  fail    // @=5 (r0,
3   or   $5,  $31,  $31     //      w1)
4   bne  $6,  $30,  fail    // @=6 (r0,
5   or   $6,  $31,  $31     //      w1)
6   bne  $7,  $30,  fail    // @=7 (r0,
7   or   $7,  $31,  $31     //      w1)
8   bne  $8,  $30,  fail    // @=8 (r0,
9   or   $8,  $31,  $31     //      w1)
10           ...
```

Les deux ports de lecture

Le banc de registres dispose de deux ports de lecture A et B (cf figure 3.1). Si l'on regarde le séquencement dans le listing 3.2, on peut constater que la séquence d'adresses se déroule sur le premier registre. Les formats d'instructions (figure 3.3) montrent que le premier registre est `Rs`. Suivant l'architecture, le champs `Rs` peut être branché au port A ou B, le champs `Rt` sur l'autre. Supposons, sans perte de généralité, que le champs `Rs` soit le port A, le champs `Rt` le port B. Pour effectuer une lecture sur le port B, il suffit de doubler chaque lecture en permutant le registre testé et le registre référence. Le listing 3.3 présente une séquence de lecture sur les deux ports.

Listing 3.3: Lecture sur les ports A et B

```
1           ...
2   bne  $5,  $30,  fail    // @=5(PORT A) (r0,
3   bne  $30, $5,   fail    // @=5(PORT B)  r0,
4   or   $5,  $31,  $31     //              w1)
5   bne  $6,  $30,  fail    // @=6(PORT A) (r0,
6   bne  $30, $6,   fail    // @=6(PORT B)  r0,
7   or   $6,  $31,  $31     //              w1)
8   bne  $7,  $30,  fail    // @=7(PORT A) (r0,
9   bne  $30, $7,   fail    // @=7(PORT B)  r0,
10  or   $7,  $31,  $31     //              w1)
11  bne  $8,  $30,  fail    // @=8(PORT A) (r0,
12  bne  $30, $8,   fail    // @=8(PORT B)  r0,
13  or   $8,  $31,  $31     //              w1)
14          ...
```

3.2.4 Architecture pipeline et "bypass"

Dans une architecture contenant un pipeline, des "bypass" (raccourcis) matériels existent afin d'éviter toute violation de dépendance de données. En effet, lorsque l'instruction $i+1$ utilise comme opérande un résultat calculé par l'instruction i qui n'a pas été encore mise à jour dans le banc de registres, l'instruction $i+1$ ne lit pas directement le banc de registres, mais utilise un bypass permettant de lire

la valeur demandée depuis un registre du pipeline. Ainsi, l'instruction $i+1$ n'a pas effectué de lecture dans le banc de registres.

Le listing 3.4 illustre l'activation de bypass entre les instructions 2 et 3 ainsi que 4 et 5. L'instruction 3 a pour registre source le même registre que le registre destination de l'instruction 2. L'instruction 3 ne va pas directement lire le banc de registres, un bypass est activé. Même chose pour l'instruction 5. Ce phénomène diminue le taux de couverture lorsque le banc de registres est testé puisque certaines instructions de lecture ne sont pas effectuées.

Listing 3.4: Activation de raccourcis matériels (bypass)

```
1          ...
2  or   $5,  $31,  $31     // @=5 (w0,
3  bne  $5,  $31,  fail     //      r0)
4  or   $6,  $31,  $31     // @=6 (w0,
5  bne  $6,  $30,  fail     //      r0)
6          ...
```

Afin d'éviter l'activation des bypass, des NOPs (No OPeration) doivent être injectés entre l'instruction productrice et l'instruction consommatrice. Le nombre d'instructions NOP à ajouter dépend principalement de la profondeur du pipeline. Le listing 3.5 présente l'introduction des instructions NOP afin d'effectuer correctement chaque opération March sur le banc de registres. Trois NOPs sont nécessaires dans le cas d'un processeur MIPS R3000 ayant un pipeline de 5 étages. L'introduction de NOPs entraîne une augmentation de la taille des programmes de test.

Listing 3.5: Injection de NOPs pour éviter l'activation des bypass

```
1          ...
2  or   $5,  $31,  $31     // @=5 (w0,
3  nop
4  nop
5  nop
6  bne  $5,  $31,  fail     //      r0)
7  or   $6,  $31,  $31     // @=6 (w0,
8  nop
9  nop
10 nop
11 bne  $6,  $30,  fail     //      r0)
12         ...
```

3.3 Partitionnement du banc de registres

Nous devons prendre en compte le fait que le programme d'auto-test doit être le plus petit possible en mémoire. L'utilisation de fonctions ou de boucles permet une réduction drastique de la taille du code. C'est pourquoi quelques registres doivent être réservés afin de contenir des variables, des compteurs, des arguments de fonctions ou bien des adresses. De plus, comme nous l'avons vu précédemment, les instructions permettant l'implémentation des primitives March requièrent des registres contenant des valeurs références ou bien des adresses. Ces registres sont alors placés dans une *zone réservée* du banc de registres ne subissant pas le test. Comment tester le banc de registres entier, alors que certains registres ne peuvent subir le test ?

Une solution est proposée dans [Paschalis04]. Elle consiste en la division du banc de registres en deux parties. Nous montrons dans cette section que cette approche est inadaptée à un banc de registres conçu comme une mémoire. Pour rappel, dans [Paschalis04] le banc de registres est vu comme une netlist de portes logiques et de bascules. C'est pourquoi nous proposons une solution, adaptée à une architecture de mémoire, qui consiste à diviser le composant en trois parties.

3.3.1 Bipartition du banc de registres et problèmes engendrés

Bipartition : test en deux phases

La solution proposée dans [Paschalis04] repose sur la division du banc de registres en deux parties. Une partie (partie A) est utilisée pour ce que nous avons appelé la *zone réservée*, où les variables, compteurs et adresses sont stockés. L'autre partie (partie B) subit le test. Lorsque le test de la partie B est achevé, les deux parties sont permutées afin de tester la partie A non encore testée, la partie B devenant la *zone réservée*. Le test du banc de registres s'effectue donc en deux étapes. La figure 3.4 décrit le partitionnement du banc de registres et les deux étapes du test.

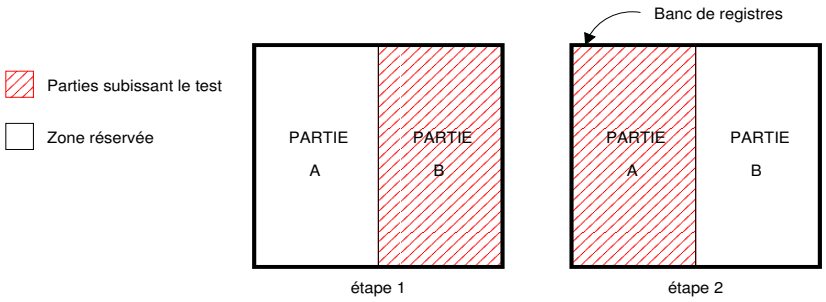


FIG. 3.4: Bipartition du banc de registres et test en deux étapes

Voyons quels sont les problèmes liés à la bipartition du banc de registres lorsque des modèles de fautes de mémoire sont utilisés.

Non détection des fautes d'adresses (AF)

Van de Goor [Goor91] classe les fautes d'adresses des mémoires orientées bit en quatre cas. La figure 3.5 illustre ces fautes. La figure 3.6 présente les combinaisons de fautes d'adresses qui doivent être testées.

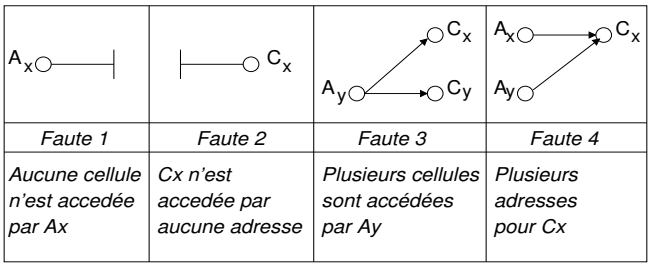


FIG. 3.5: Fautes du décodeur d'adresses

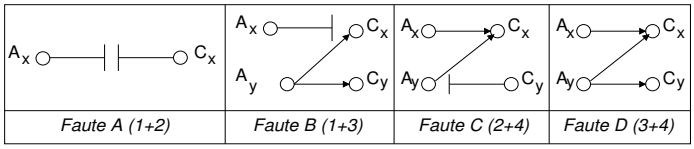


FIG. 3.6: Combinaison des fautes d'adresses à détecter

Enfin, le tableau 3.1 présente les deux conditions nécessaire et suffisante pour détecter toutes les fautes d'adresses. Les '...' dans les éléments March représentent n'importe quel nombre de lecture et d'écriture. Avant la condition 1, un élément March de type $\Updownarrow (wx)$ doit être appliqué. La valeur de x peut être choisie librement entre 0 et 1, mais doit rester la même tout au long du test.

Condition	Élément March	Condition	Élément March
1	$\Uparrow (rx, ..., w\bar{x})$	2	$\Downarrow (r\bar{x}, ..., wx)$

TAB. 3.1: Condition pour la détection des fautes d'adresses.

En ayant une simple bipartition du banc de registres, nous pouvons constater que les fautes d'adresses impliquant au moins deux cellules, appartenant à une partie différente du banc de registres, ne peuvent être détectées. En effet, prenons l'exemple de la faute D de la figure 3.6. Supposons que A_x désigne le registre `$X` appartenant à la partie A du banc de registres, et A_y, le registre `$Y` situé dans la partie B. Le test est effectué sur la partie B du banc de registres. Supposons un test "normal" de la mémoire. La mémoire est initialisée à 0. Lors de l'écriture d'un 1 à A_y, C_y prend la valeur 1 ainsi que C_x (faute D). A la lecture d'A_x nous obtenons un 1 au lieu d'un 0. La faute est détectée. Mais, si le test n'est effectué que sur la partie B (contenant `$Y`) la lecture à l'adresse A_x n'est pas effectuée et la faute n'est pas détectée.

Non détection des fautes de couplage (CF)

Certains algorithmes March ont été conçus afin de détecter les fautes de couplage. Ainsi le March C- peut détecter une faute de couplage entre la cellule C_i dans le registre `$X` et la cellule C_j dans le registre `$Y`. De la même façon que pour les fautes d'adresses impliquant deux cellules, si les registres `$X` et `$Y` sont localisés dans la même partie (A ou B) du banc de registres alors la faute de couplage peut être détectée. Dans le cas contraire, `$X` se situant dans la partie A et `$Y` dans la partie B, puisqu'aucun test n'est exécuté entre ces deux registres en même temps, la faute de couplage ne pourra être détectée par le test.

3.3.2 Tripartition du banc de registres et problèmes résolus

Tripartition : test en trois phases

Le bi-partitionnement du banc de registres ne permet pas de détecter les fautes entre deux cellules n'appartenant pas à la même partie. Pour y remédier, nous proposons de tri-partitionner le banc de registres et d'effectuer le test en trois étapes. A chaque étape, deux parties sont testées en même temps, la dernière étant utilisée comme *zone réservée*. La figure 3.7 illustre ce test en 3 étapes. Lors de la première étape les parties A et B sont testées. La partie C est alors utilisée comme *zone réservée*. Les parties B et C sont testées dans un second temps, la partie A étant la *zone réservée*. Enfin, la troisième et dernière étape permet de tester les parties A et C ensemble.

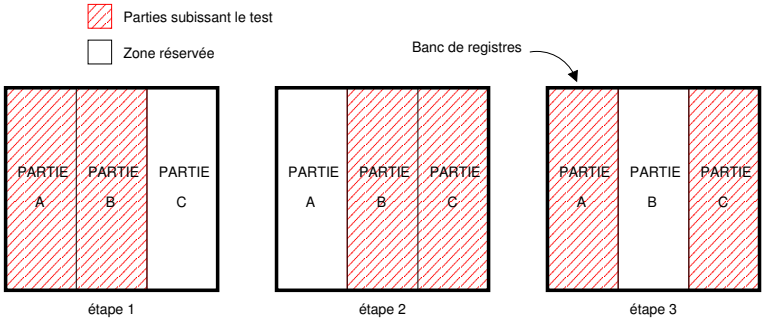


FIG. 3.7: Tripartition du banc de registres et test en trois étapes

Détection des fautes d'adresses et de couplage

Le problème dans la détection de ce type de fautes dans le cas d'un partitionnement de la mémoire se situe dans la détection des fautes inter-parties. La tripartition et le test en trois étapes proposé ci-dessus résolvent le problème ainsi : n'importe quelle faute impliquant deux parties se trouve soit :

- dans la partie A-B : détectée durant l'étape de test 1
- dans la partie B-C : détectée durant l'étape de test 2
- dans la partie A-C : détectée durant l'étape de test 3

Si une faute implique les trois parties (faute d'adresses par exemple), ceci signifie qu'elle implique au moins deux parties. Elle sera donc détectée. Un désavantage de la tripartition se situe dans la

redondance des tests. En effet, chaque partie est testée deux fois, augmentant le temps de test. Ainsi, un algorithme en $O(n)$ passera en $O(2n)$.

3.4 La plate-forme de simulations

Afin de valider notre méthode et de faire les mesures correspondantes, une plate-forme de simulation en SystemC à été réalisée (figure 3.8). La simulation est précise au cycle près et au bit près, afin de mesurer avec exactitude le nombre de cycles nécessaires à l'exécution complète du programme de test, ainsi que sa taille exacte en mémoire. Le microprocesseur utilisé est un MIPS R3000 avec un pipeline de 5 étages. Le banc de registres du microprocesseur contient 32 registres de 32 bits comme décrit dans la section 3.1.2. Le microprocesseur est connecté à une mémoire cache. Un interconnect, plus exactement un cross-bar, respectant la norme VCI [VSIA] est utilisé pour connecter l'ensemble microprocesseur et son cache avec une mémoire. Cette dernière contient le binaire qui provient de l'assemblage du programme de test. Les différents composants SystemC proviennent de la bibliothèque de composants SOCLIB [SoCLIB]. Le processus de test se déroule en trois étapes :

1. **Les programmes de test.** Les programmes de test sont générés automatiquement par un outil. Cet outil prend en entrée un fichier décrivant l'algorithme March ainsi que les mots de fond utilisés. Il génère alors le programme d'auto-test correspondant en assembleur.
2. **Les fichiers de traces.** Lors de la simulation, le banc de registres génère un fichier de traces contenant l'ensemble des opérations de lecture et d'écriture subis.
3. **La simulation de fautes.** Afin de vérifier que les taux de couverture théoriques sont atteints, un simulateur de fautes de mémoire (RAMSES) est utilisé. Il prend en entrée le fichier de traces et renvoie les taux de couverture obtenus.

La suite présente de façon plus détaillée ces trois points.

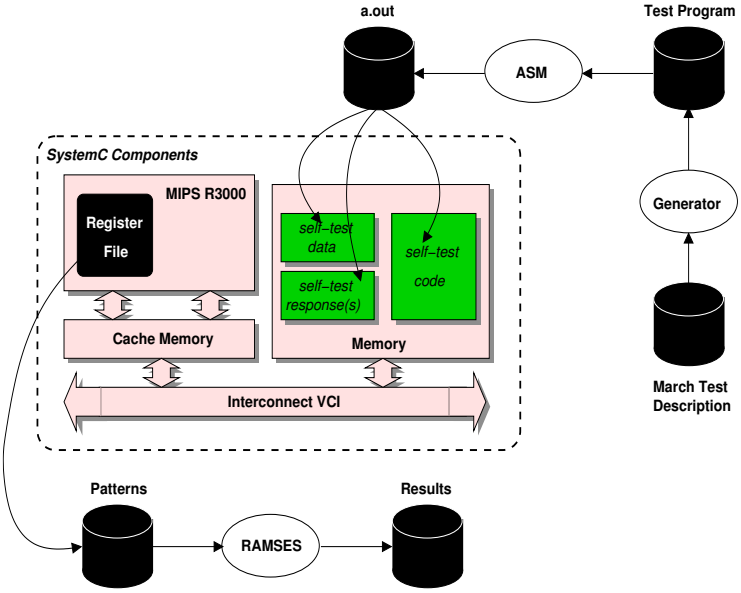


FIG. 3.8: Plate-forme de test du banc de registres

3.4.1 Génération automatique des programmes de test

L'approche systématique que nous avons employée nous a permis d'automatiser la génération des programmes de test à partir d'une description d'une séquence d'éléments March. La figure 3.9 illustre les entrées/sorties de l'outil. Le générateur prend en entrée une description d'un test March sous forme d'une séquence d'éléments March (ligne commençant par `e`). Les lignes commençant par `bg` donnent les mots de fond sur lesquels l'algorithme doit être itéré. Le programme assembleur donné en sortie respecte la tripartition (section 3.3.2) ainsi que l'injection de NOPs nécessaires à la non activation des raccourcis dans le pipeline (section 3.2.4). Le générateur nous permet ainsi de tester n'importe quelle séquence March avec n'importe quelle succession de mots de fond.

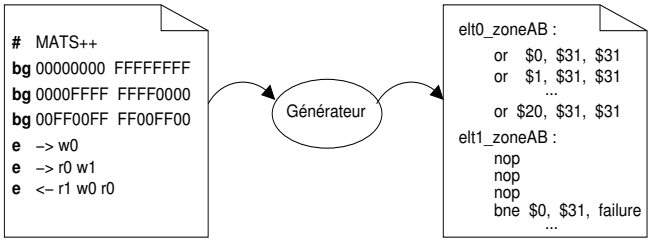


FIG. 3.9: Génération automatique des programmes de test du banc de registres

3.4.2 Le simulateur de fautes : RAMSES

Afin d'analyser et de vérifier les taux de couverture obtenus grâce aux programmes de test, un simulateur de fautes a été utilisé sur les fichiers de traces générés par le banc de registres durant la simulation. Le simulateur de fautes de mémoire RAMSES [Wu99], mis au point à l'université de Tsing Hua (Taiwan) a été utilisé. L'interface de RAMSES a été légèrement modifiée (pas le moteur de simulation) afin qu'il puisse directement prendre en entrée les fichiers de traces générés par notre composant SystemC. La figure 3.10 résume l'utilisation de RAMSES dans son comportement modifié. Les détails peuvent être trouvés dans le rapport [Garcia06].

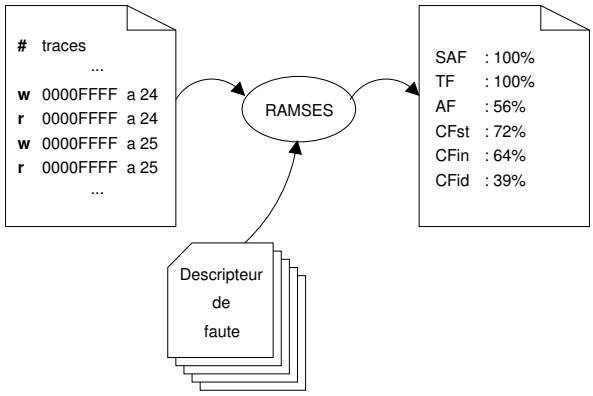


FIG. 3.10: RAMSES dans sa version modifiée prend en entrée un fichier de traces

3.5 Résultats expérimentaux

Différents programmes de test ont été générés. Chaque programme reflète l'exécution d'un test March différent. Les cinq algorithmes MATS, MATS+, MATS++, MARCH X et MARCH C- (voir tableau 2.1) ont été utilisés pour le test du banc de registres. Ces cinq algorithmes vont en complexité croissante et en qualité de test croissante. Ainsi les résultats obtenus permettront d'effectuer des choix en fonction du compromis temps de test/qualité de test. Pour chaque test March deux versions ont été réalisées. La première version n'exécute le test que pour un unique couple de mots de fond (le numéro 1 du tableau 2.3). La deuxième version exécute le test March successivement pour les 6 couples de mots de fond du tableau 2.3. Ainsi, les résultats sont rapportés pour l'exécution de dix programmes de test sur le banc de registres.

3.5.1 Taux de couverture

Le tableau 3.2 présente les taux de couverture calculés par RAMSES. La première colonne présente l'algorithme March utilisé. La deuxième colonne précise la version du test March effectué. Lorsque DB=1, le test est exécuté pour un couple de mot de fond. Lorsque DB=1–6, le test est réitéré sur les six couples. Les colonnes suivantes présentent les taux de couverture obtenus.

Test	DB	SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
MATS	1	100.0 %	100.0 %	59.8 %	69.7 %	69.7 %	37.5 %
	1–6	100.0 %	100.0 %	98.9 %	99.4 %	99.4 %	90.4 %
MATS+	1	100.0 %	100.0 %	96.9 %	69.5 %	80.3 %	44.8 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	99.4 %	92.6 %
MATS++	1	100.0 %	100.0 %	96.9 %	69.5 %	81.5 %	45.4 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	99.4 %	92.8 %
MARCH X	1	100.0 %	100.0 %	96.9 %	73.2 %	100.0 %	55.5 %
	1–6	100.0 %	100.0 %	100.0 %	99.4 %	100.0 %	99.3 %
MARCH C-	1	100.0 %	100.0 %	96.9 %	98.4 %	100.0 %	98.4 %
	1–6	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %

TAB. 3.2: Taux de couverture obtenus après simulation par les programmes auto-testants.

Les résultats sont ceux escomptés (cf tableau 2.2), excepté pour la détection des fautes de transition (TF) qui se révèle être plus élevée dans certains cas. En effet, RAMSES nous indique que les fautes de transition sont détectées à 100% par des algorithmes qui ne sont pas sensés les detecter (MATS, MATS+). Cette détection est en fait du à la tripartition du banc de registres, plus particulièrement à la redondance du test. Comme nous l'avons vu dans la section 3.3.2, chaque partie est testée deux fois par le même algorithme. Or, un MATS(+) répété deux fois de façon identique sur la même partie de la mémoire est en mesure de détecter les fautes de transition.

Les algorithmes permettant de détecter les fautes d'adresses AF (MATS+, MATS++, MARCH X et MARCH C-), lorsqu'ils ne sont exécutés qu'avec un unique mot de fond, ne peuvent détecter les

fautes d'adresse intra-mot. Ce qui explique le taux de couverture des AFs de 96.9% pour ces tests lorsque DB=1. Ce taux passe à 100% lorsque l'algorithme est répété sur les 6 mots de fond.

3.5.2 Temps de test

Le tableau 3.3 reporte le nombre de cycles nécessaires à l'exécution des programmes de test. Ces chiffres représentent le cas d'une mémoire parfaite répondant en un cycle. La colonne 2 donne le nombre de cycles nécessaires à l'exécution d'un test pour 1 DB, la colonne 4 pour 6 DBs. Nous pouvons constater, à la comparaison de ces deux colonnes, que l'utilisation des 6 couples de mots de fond, bien qu'augmentant le taux de couverture, sont **6 fois** plus long à l'exécution.

Test	# Cycles (1 DB)	$\frac{\#Cycles\ Gizo}{\#Cycles(1DB)}$	# Cycles (6 DBs)	$\frac{\#Cycles\ Gizo}{\#Cycles(6DBs)}$
MATS	674	0.78	3,944	4.59
MATS+	921	1.07	5,426	6.31
MATS++	1,058	1.23	6,248	7.27
MARCH X	1,058	1.23	6,248	7.27
MARCH C-	1,826	2.12	10,856	12.63

TAB. 3.3: Nombre de cycles nécessaires à l'application de programmes de test dans le cas d'une mémoire parfaite.

Comparaison avec l'approche proposée dans [Gizopoulos04]

Il est intéressant de comparer ces résultats avec ceux obtenus par l'approche structurelle décrite dans [Gizopoulos04]. En effet, comme nous l'avons noté précédemment, la technique décrite dans ce livre, puisque bas niveau, donne des programmes de test petits et rapides à l'exécution. Nous prendrons comme référence l'étude de cas du Meister/MIPS qui est un MIPS R3000 pipeline à 5 étages. Le banc de registres est le même que celui étudié, à savoir, un banc de registres de 32 registres de 32 bits, avec un port d'écriture et deux ports de lecture. La méthode de test employée est décrite dans la section 3.1.2. Le nombre de cycles nécessaires est de **859** cycles. La colonne 3 du tableau 3.3 rapporte le ratio entre ce nombre (appelé #Cycles Gizo) et le nombre de cycles lorsque DB=1 (colonne 2). La colonne 5, présente le ratio entre le nombre de cycles obtenus par la méthode structurelle et la colonne 4. Les chiffres de la colonne 3 montrent que, lorsque les tests March sont exécutés avec un seul mot de fond, le temps de test est a peu près équivalent à l'approche structurelle. En effet la moyenne est de **1.3**.

Il faut se rappeler cependant, que dans le cas du MATS+ version 1 DB, bien que nécessitant le même nombre de cycles que l'approche structurelle, MATS+ garantie la détection de 100% des SAFs et TFs, ainsi que 97% des AFs et environ 65% des CFs en moyenne (cf tableau 3.2). L'approche proposée dans [Gizopoulos04] n'est prévu que pour les détections des SAFs.

Comme nous l'avons constaté, les versions 6 DBs sont plus longues à l'exécution. En moyenne, ces versions sont **7.6** fois plus longues que l'approche structurelle. Ces résultats se basent sur l'utilisation d'une mémoire parfaite répondant en un cycle. Or dans un contexte système, lorsque le programme est chargé dans la mémoire principale, les effets dûs aux caches du processeur doivent être pris en compte. Il est donc intéressant d'étudier l'influence des caches sur les temps de test.

Influence des caches

Le tableau 3.4 reporte le nombre de cycles nécessaires à l'exécution des programmes de test dans un contexte système où le processeur est muni d'un cache de 8Ko (4Ko pour le cache instructions et 4Ko pour le cache de données). Bien évidemment on peut constater que, dans ce contexte, les temps de test sont plus longs. Néanmoins, il est intéressant de noter que la différence entre l'exécution du test dans sa version 1 DB et 6 DB s'est réduite. La dernière colonne du tableau 3.4 présente le rapport entre ces deux versions. Pour un cache instructions de 4Ko le rapport entre le temps de test de la version 6 DB et le temps de test de la version 1 DB est de seulement **2.76** en moyenne. Il était de 6 dans le cas d'une mémoire parfaite.

On peut distinguer sur la figure 3.11 le rapport entre les versions 6 DBs et 1 DB de chaque algorithme pour différentes tailles du cache instructions. Plus la taille du cache instructions augmente, plus la différence en temps de test entre les deux versions s'estompe.

En effet, prenons la courbe du test March C- (figure 3.11, carrés vides). Le rapport entre la version 6 DBs et 1 DB est de **5.9** pour un cache instructions de 1Ko. Ce qui signifie que la version 6 DBs met environ 6 fois plus de temps à s'exécuter que la version 1 DB. Ce rapport passe à **2.5** pour un cache instructions de 16Ko.

Test	# Cycles (DB 1)	# Cycles (DB 1–6)	$\frac{\#Cycles\ (DB\ 1-6)}{\#Cycles(DB\ 1)}$
MATS	2,061	5,556	2.7
MATS+	2,773	7,518	2.71
MATS++	3,180	8,955	2.82
MARCH X	3,180	8,955	2.82
MARCH C-	5,478	15,228	2.78

TAB. 3.4: Nombre de cycles nécessaires à l'application de programmes de test dans le cas d'un processeur muni d'un cache instructions de 4Ko.

Pour résumer, les programmes de test March en version 1 DB ne sont, en moyenne, que **1.3** fois plus longs que le programme implémentant la version de test structurelle. Et, en prenant en compte les effets des mémoires caches du processeur, le rapport entre les versions 6 DBs et les versions 1 DB tend vers **2**.

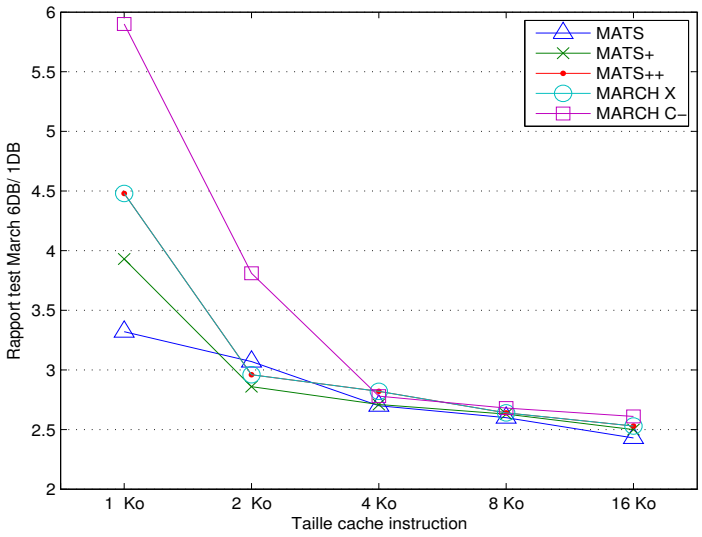


FIG. 3.11: Rapport du temps d'exécution entre la version 6 DBs et 1 DB de chaque algorithme March en fonction de la taille du cache instructions.

3.5.3 Taille des programmes de test

Comme nous l'avons précisé plus tôt, selon [Gizopoulos04], le temps de test global est dominé par le chargement du programme de test dans le cas où la fréquence de fonctionnement du SoC est plus élevée que celle du testeur externe. La taille des programmes de test est donc un facteur important du temps global de test.

Le tableau 3.5 reporte la taille des programmes de test en Kilo-octets (Ko). La colonne 1 (resp. 3) présente la taille des programmes utilisant 1 DB (resp. 6 DBs). La colonne 2 (resp. 4) présente le ratio entre la taille du programme de test structurel (appelé Gizo dans le tableau) et la taille des programmes implémentant le test pour 1 DB (resp. 6).

Dans [Gizopoulos04], le test pris en référence (Meister/MIPS R3000), compte **720** mots soit **2.8**Ko. Le ratio se limite entre 0.92 et 2.70 fois la taille du programme de test structurel de Gizopoulos. La moyenne est de **1.6**.

Test	Taille pour 1 DB	$\frac{Taille\ Gizo}{Taille\ DB\ 1}$	Taille pour 6 DBs	$\frac{Taille\ Gizo}{Taille\ DB\ 1-6}$
MATS	2.59 Ko	0.92	3.06 Ko	1.09
MATS+	3.56 Ko	1.27	4.03 Ko	1.43
MATS++	4.09 Ko	1.46	4.56 Ko	1.62
MARCH X	4.09 Ko	1.46	4.56 Ko	1.62
MARCH C-	7.09 Ko	2.53	7.56 Ko	2.70

TAB. 3.5: Taille des programmes de test

On peut remarquer que l'augmentation de la taille du code du passage de la version 1 DB à la version 6 DBs n'est pas exorbitante. En effet, en moyenne, les versions 6 DBs ne sont que **1.12** fois plus grosses que les versions 1 DB. Ce qui signifie que pour une augmentation de la taille du programme de test de seulement **12%**, le test est de meilleure qualité (cf tableau 3.2).

3.6 Conclusion

Bien qu'étant un composant crucial des processeurs à architecture RISC, le banc de registres reste peu étudié dans la littérature concernant l'auto-test logiciel. Une première approche détaillée dans [Kranitis03a] et [Gizopoulos04] considère le banc de registres comme une netlist de portes logiques. Mais, dans la plupart des cas, le banc de registres est conçu comme une petite mémoire SRAM. Une deuxième approche [Goor92] propose donc d'utiliser les algorithmes March dans les programmes d'auto-test pour tester le banc de registres. Malheureusement, aucune indication, ni sur la méthode employée ni sur les résultats n'est dévoilée.

Nous avons donc présenté une méthode dont les caractéristiques sont :

- Une approche systématique pour la génération des primitives March.
- Une gestion des erreurs producteurs/consommateurs provoquées par les architectures pipelines.
- La nécessité d'utiliser une zone réservée au sein du banc de registres ne pouvant subir le test :
    - Nous avons montré que la bipartition du banc de registres ne permettait pas la détection de toutes les classes de fautes.
    - Nous avons proposé une solution basée sur la tripartition du banc de registres et un test en 3 étapes.

Les résultats obtenus ont été comparés à l'approche structurelle proposée dans [Gizopoulos04]. Cette dernière approche, pour rappel, ne permet la détection que des SAFs. A un bout du spectre nous avons le test MATS (en version 1 DB) qui se révèle être plus léger et plus rapide. A l'autre bout du spectre, le test MARCH C- (en version 6 DBs) est 6 fois plus long à l'exécution (effets des caches pris en compte) et 2.7 fois plus gros. Mais ce dernier test garantit une détection à 100% de tous les modèles de fautes étudiés. Il s'agit donc de faire un compromis entre le coût du test et sa qualité.

Chapitre 4

Auto Test Logiciel des Mémoires Caches

Après avoir étudié l'auto-test logiciel des bancs de registres, ce chapitre s'attaque à l'étude du test des mémoires caches. Le test des mémoires caches est différent de celui des bancs de registres. Autant le banc de registres peut être adressé directement par le langage d'assemblage, autant les mémoires caches fournissent des mécanismes d'accélération "invisibles" au programmeur. Le test logiciel des mémoires caches ne peut donc s'effectuer que par effets de bord.

Ce chapitre, comme le précédent, est divisé en trois parties principales. La première partie a pour but de préciser l'architecture des mémoires caches et présente l'état de l'art.

La deuxième partie de ce chapitre débute par une présentation de l'architecture du cache étudié. En effet, le programme de test doit être conçu en fonction des caractéristiques du cache afin d'anticiper son comportement dynamique. Ensuite nous présenterons les stratégies de test à proprement parler. Dans un premier temps le cas du cache de données sera abordé, dans un second temps, nous nous focaliserons sur le cache instructions.

La troisième et dernière partie présentera la plate-forme de simulation. Nous présenterons la façon dont nous avons validé notre approche et exposerons les résultats obtenus, sur le cache de données puis sur le cache instructions. Ces résultats seront comparés avec une approche BIST partagée.

Finalement, une dernière section conclura ce chapitre.

4.1 État de l'art

Avant de passer en revue l'état de l'art sur l'auto-test logiciel des mémoires caches, nous présentons les caractéristiques principales de ces mémoires ainsi que le vocabulaire associé.

4.1.1 Architecture des mémoires caches

Une mémoire cache est une petite mémoire rapide, physiquement proche du microprocesseur. Elle contient des copies des données de la mémoire principale les plus fréquemment utilisées. Ainsi, l'accès à ces données cachées permet de réduire la latence moyenne d'accès à la mémoire. Les caches s'appuient sur les caractéristiques de localité des programmes :

- localité spatiale : une donnée, voisine d'une autre qui vient d'être accédée, a une forte probabilité d'être accédée prochainement. En effet, les programmes sont rangés en séquence, les données en tableaux.
- localité temporelle : une donnée qui vient d'être accédée a une forte probabilité d'être accédée prochainement. En effet, les programmes contiennent des boucles, les données sont accédées de façon répétée (ex : les compteurs).

Les échanges entre la mémoire et le cache se font avec une granularité de *bloc/ligne* de cache : un nombre N d'octets consécutifs, puissance de 2, alignés en mémoire. La figure 4.1 présente la division de la mémoire principale en bloc de N octets, ainsi que la structure générale d'une mémoire cache. La partie Tag de la mémoire cache contient le numéro du bloc présent dans la ligne du cache.

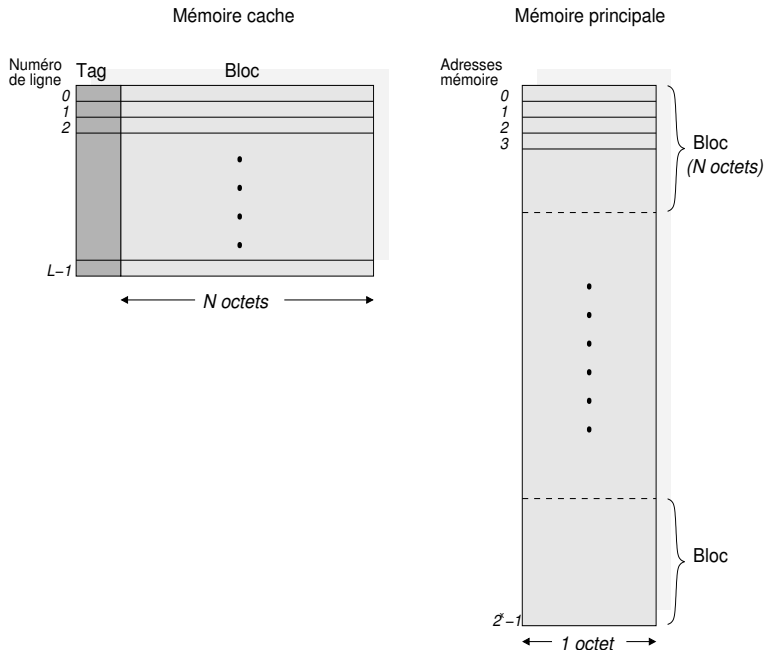


FIG. 4.1: Structure de la mémoire principale et de la mémoire cache

Associativité

L'associativité détermine le nombre de lignes dans le cache, dont dispose chaque bloc de la mémoire principale. Il existe trois stratégies d'associativité :

- Cache totalement associatif (fully associative) : tous les blocs de la mémoire peuvent être rangés dans n'importe quelles lignes du cache.
- Cache à correspondance directe (direct mapping) : l'ensemble des blocs de la mémoire sont partitionnés en M familles. A chaque famille, une unique ligne du cache est allouée. L'appartenance à une famille (numéro de famille) est définie par les m bits ($M = 2^m$) de poids faible du numéro de bloc.
- Cache partiellement associatif (n-way set associative) : à chaque famille est alloué un nombre de lignes (2, 4, 8 ou 16) qui sont équivalentes.

Dans le cas des caches associatifs (partiellement ou totalement), la sélection d'une ligne parmi n doit être effectuée. Plusieurs stratégies peuvent être envisagées : RANDOM, FIFO (First-In First-Out), LRU (Least-Recently Used).

Politiques des écritures en mémoire principale

Quand une donnée se situe dans le cache, le système en possède deux copies : une dans la mémoire principale et une dans la mémoire cache. La cohérence mémoire mesure la différence entre les données de la mémoire principale et les données dans le cache. Il existe deux stratégies pour les écritures en mémoire principale :

- Write-Through : écriture systématique dans la mémoire. Le cache n'est mis à jour que si la donnée est déjà présente dans le cache.
- Write-Back : les écritures se font dans le cache et non dans la mémoire. La mémoire n'est mise à jour par le cache que si l'on réquisitionne une ligne du cache contenant des données modifiées.

Afin d'éviter le sur-coût dû aux écritures (lorsque le cache met à jour la mémoire principale), le cache dispose d'un tampon d'écriture (Write-Buffer). Le Write-Buffer est en général une petite mémoire FIFO. Les écritures en mémoire principale se font ainsi de façon groupée.

Après cette description du fonctionnement et de l'architecture des mémoires caches, passons à l'étude de l'état de l'art. L'auto-test logiciel des mémoires caches reste peu étudié dans la littérature. Deux articles traitent du sujet. Le premier s'applique au test des caches du microprocesseur Intel i860. Le deuxième traite des caches du microprocesseur Alpha AXP 21164.

4.1.2 Test des caches du processeur Intel i860^{TM}

Le test du processeur RISC Intel i860^{TM} proposé dans [Goor92] repose sur une méthodologie logicielle. Van de Goor et Verhallen étendent le modèle fonctionnel présenté dans [Thatte80, Brahme84] à la classe des mémoires. Nous nous intéresserons dans cette section à la partie de l'article traitant du test de la mémoire cache.

Le processeur i860 possède une mémoire cache instructions et données associatives à deux voies. Le cache de données a une taille de 8Ko et adopte une politique d'écriture postée de type "Write-Back". Le cache instructions a une taille de 4Ko.

Test du cache de données

La figure 4.2 présente les deux types de mémoire contenus dans le cache de données : la mémoire dans laquelle les données effectives sont rangées (Data Memory Array sur la figure) et la mémoire contenant les numéros des blocs présents dans le cache (Directory Memory Array). Par la suite nous nous intéresserons qu'à la première famille de mémoire qui représente plus de 90% de la mémoire totale du cache de données.

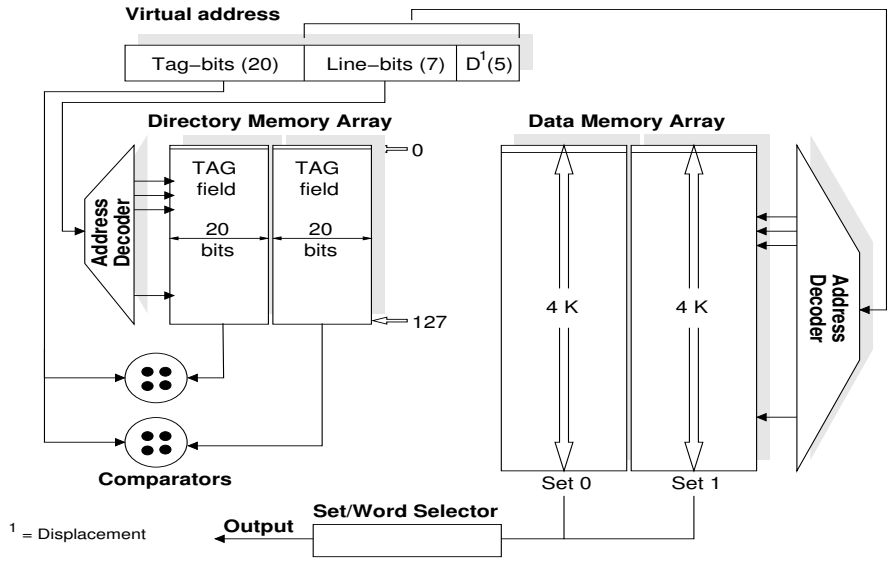


FIG. 4.2: Structure interne du cache de données de l'Intel i860

Pour le test de la mémoire contenant les données, il est précisé dans l'article que n'importe quel test de mémoire standard peut être utilisé. Afin de couvrir les modèles de fautes indiqués dans la section 3.1.2 le test March B $\{ \Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0) \}$ [Goor91] est utilisé. Le cache est 2-voies associatif, l'algorithme de sélection de la ligne utilise une technique pseudo-aléatoire par LFSR [Rhodehamel89]. La voie qui sera utilisée pendant le déroulement du programme reste donc inconnue. Pour simplifier le test, un registre du processeur appelé DIRBASE, permet de sélectionner la voie utilisée. Ainsi, pour le test, le cache peut

être manipulé de la même façon qu'un cache à correspondance directe. Le test est alors effectué une première fois sur une voie, et une seconde fois sur la deuxième.

Test du cache instructions

Pour la même raison que pour le cache de données nous restreignons notre étude à la mémoire contenant les données effectives (les instructions dans ce cas). A la différence du cache de données, le cache instructions ne peut contenir n'importe quelles données. En effet, ne sont contenues que des instructions valides. A cause des possibilités limitées d'application de constantes immédiates dans le jeu d'instruction, le '0' et le '1' du test March B doivent être remplacés par :

ZERRO : une séquence d'instructions qui effectue une transformation unique d'un ou plusieurs registres et satisfait la condition qu'un ET logique de cette séquence d'instructions soit 000..0 (tout à 0).
ONNE : une séquence d'instructions qui effectue une transformation unique d'un ou plusieurs registres et satisfait la condition qu'un OU logique de cette séquence d'instructions soit 111..1 (tout à 1).

L'exécution des instructions ne permet de lire cette mémoire que dans un ordre ascendant. Ainsi seule les fautes de collage et les fautes de transition peuvent être détectées. Le test de fautes plus complexe comme les fautes de couplage ne peut être garanti.

Les problèmes liés à cette approche sont :
– Aucun renseignement n'est donné sur l'implémentation du test.
– Aucune valeur numérique sur la taille des programmes générés ou le nombre de cycles nécessaires à leur exécution n'est donnée.

4.1.3 Test des caches du processeur Alpha AXP 21164

L'article [Bhavsar94] traite de l'auto-test logiciel des caches du processeur Alpha AXP 21164. Bien que l'article aborde la totalité de l'aspect testabilité du microprocesseur, nous nous intéresserons qu'à la partie consacrée au test des mémoires caches.

En ce qui concerne le test des mémoires embarquées, le BIST des mémoires à été clairement le choix. Cependant, forcer cette solution à toutes les RAMs aurait fait porter un énorme fardeau sur les ressources de conception. Pour le processeur Alpha 21164, ce problème a été résolu comme suit : le cache instruction a été identifié comme le noyau clé du test sur la puce. Ainsi, il est testé en utilisant un BIST [Dekker88a] implémentant un algorithme March 9N [Dekker88b]. Un mécanisme de type BISR est aussi présent. Après son auto-test (et son éventuelle réparation) un noyau connu fonctionnel est disponible. Ensuite, un programme d'auto-test peut être chargé dans le cache instruction et permet de tester le cache de données.

Le cache du processeur Alpha AXP 21164 est un cache de 2x8Ko (8Ko cache instructions, 8Ko cache de données). Le cache est à correspondance directe, la politique d'écriture est "Write-Through".

Les problèmes liés à cette approche sont :
– Aucune indication n'est donnée sur l'implémentation du test du cache de données.
– Aucune mesure de la qualité du test n'est donnée puisqu'aucun algorithme de test du cache de données n'est présenté.
– Aucun chiffre n'est donné sur la taille ou le nombre de cycles nécessaires au logiciel de test.

Le tableau 4.1 résume les caractéristiques des mémoires caches étudiées. Il est nécessaire de rappeler que pour le test du cache de l'Intel i860, le cache 2-voies associatif se transforme en cache à correspondance directe du à la sélection de la voie lors du test.

Processeur	Intel i860		Alpha AXP 21164	
Cache	Instruction	Données	Instruction	Données
Associativité	2 voies associatif	2 voies associatif	correspondance directe	correspondance directe
Politique	-	Write-Back	-	Write-Through
Taille	4 Ko	8 Ko	8 Ko	8 Ko
Test	SBST	SBST	MBIST	SBST

TAB. 4.1: Caractéristiques des mémoires caches pour lesquels un test logiciel a été effectué.

4.1.4 Conclusion

Dans un contexte d'utilisation d'équipement de test bas coût, la fréquence de fonctionnement du testeur est largement inférieure à celle de la puce. Ainsi, le chargement du programme de test dans la puce domine le temps total du test [Gizopoulos04]. Sans connaître la taille des programmes générés, ni même leur temps d'exécution, comment savoir si l'auto-test logiciel peut rivaliser, ou simplement être une alternative viable aux techniques comme le BIST matériel pour mémoire ?

Nous proposons donc dans la suite de ce chapitre, une étude complète sur l'auto-test logiciel des mémoires caches. Dans un premier temps nous proposons une stratégie détaillée du test, sur les caches de données d'abord, puis sur les caches instructions. Une plate-forme a été développée, permettant de faire varier la taille des caches, afin d'observer le comportement des programmes de test. Finalement les résultats obtenus sont comparés à une approche BIST matériel partagé, permettant de déterminer dans quelle mesure une approche SBST est plus avantageuse qu'une approche BIST.

4.2 Auto-test logiciel des mémoires caches

Dans cette section nous allons aborder dans un premier temps, la génération des programmes de test du cache de données, puis celle du cache instructions.

Le test des mémoires caches par exécution d'un programme est subtil puisque celles-ci ne peuvent être accédées de façon directe. L'unique stratégie pour tester de façon logicielle un cache est par effet de bord. En effet, puisque le cache consiste en des copies de fragments des données contenues en mémoire principale, l'idée est de faire exécuter par le processeur embarqué un programme soigneusement rangé en mémoire principale. L'accès à des adresses précises en mémoire principale provoque la lecture ou l'écriture à des endroits précis de la mémoire cache. C'est pourquoi les programmes auto-testants des mémoires caches doivent être taillés selon les caractéristiques propres de celles-ci.

Ainsi, avant de passer à la génération des programmes de test voyons l'architecture de notre mémoire cache cible.

4.2.1 Architecture du cache étudié

Nous allons effectuer notre étude sur une mémoire cache "simple". Voici les caractéristiques de l'architecture de notre mémoire cache :

- Cache à correspondance directe.
- Politique d'écriture en mémoire "Write-Through".

Le cache étudié est divisé en différents composants comme le montre la figure 4.3.

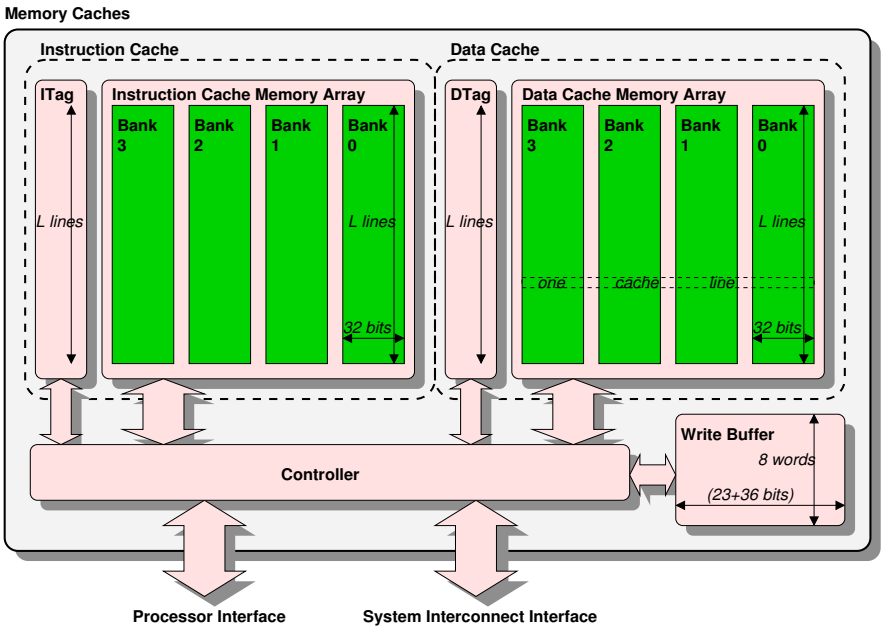


FIG. 4.3: Architecture du cache étudié

- La mémoire contenant les données du cache instruction (Instruction Cache Memory Array sur la figure, appelé **ICache** par la suite) est organisée en C bancs de RAM de L mots de 32 bits physiquement placés côte à côte. La taille du ICache est L*C*32 bits (la figure 4.3 présente un cache à 4 bancs soit $C = 4$). C représente le nombre de colonnes du cache, et L représente le nombre de lignes. Ainsi, un bloc de la mémoire principale est constitué de C mots de 32 bits consécutifs. Un bloc est contenu dans une ligne du ICache (comme présenté dans la figure 4.3).
- Le Tag du cache instruction (ITag) contient pour chaque ligne du ICache le numéro du bloc présent ainsi qu'un bit de validité.
- La mémoire contenant les données du cache de données (appelé **DCache** par la suite) a la même structure que le ICache.
- Le Tag du cache de données (DTag) a la même structure que le ITag.
- Le tampon d'écritures postées (Write Buffer) est utilisé pour l'implémentation de la politique d'écriture 'Write-Through'. Le tampon peut stocker 8 requêtes d'écriture (32 bits de donnée plus 36 bits d'adresse et de type).
- Le bloc "controller" contient les différentes machines d'état finies (FSM) permettant la communication avec le microprocesseur d'un coté, et l'interconnect système de l'autre.

4.2.2 Programme auto-testant le DCache

Comme pour le banc de registres, les programmes de test sont implémentés en langage d'assemblage MIPS (cf section 3.2.1). Dans un premier temps nous nous attarderons sur les façons d'implémenter les primitives d'écriture et de lecture March sur le DCache. Nous verrons ensuite comment gérer le séquencement des adresses. Afin de clarifier le processus, un exemple sera donné à travers l'implémentation de l'algorithme MATS.

Implémentation des tests March

Primitive d'écriture. Afin de provoquer une opération d'écriture dans le DCache, nous devons accéder à une adresse en mémoire qui n'est pas présente (ou non valide) dans le DTag (**MISS**). Ainsi, le cache ramènera de la mémoire principale le bloc correspondant. Les C bancs mémoires seront alors mis à jour. Si la mémoire principale a été correctement initialisée avec le mot de fond a, une opération wa a alors été effectuée sur chacun des C bancs. Par exemple, une instruction de type `lw` (load word) peut être utilisée afin d'effectuer l'opération précédemment décrite : `lw $5, 0($12)` (i.e. charge le mot rangé à l'adresse mémoire `$12 + 0` dans le registre `$5`). Si le registre `$12` contient une adresse non encore accédée, correspondant à ligne 0 du DCache, l'instruction provoquera l'écriture de C mots consécutifs de la mémoire principale dans les C bancs du cache. Si on incrémente la valeur de `$12` de C*4 (4 car les adresses pointent sur des octets), ré-exécuter l'instruction `lw $5, 0($12)` chargera les C mots consécutifs suivants dans la ligne 1 du DCache. Et ainsi de suite.

Primitive de lecture. Afin de provoquer une opération de lecture dans le DCache, nous devons accéder à une adresse présente et valide dans le DTag (**HIT**). Comme présenté pour le banc de registres, l'opération de lecture implique une lecture et une comparaison. Dans le cas du DCache deux instructions sont nécessaires. La première instruction charge la valeur depuis le DCache vers un registre du processeur (disons `$t`). Ensuite l'instruction `bne` peut être utilisée afin d'effectuer la comparaison entre le registre `$t` et le registre `$Ref` préalablement initialisé avec le mot de fond attendu.

Séquencement des adresses. Le cache étudié est à correspondance directe. Un bloc de la mémoire principale ne peut rentrer que dans une ligne du DCache. Ainsi, afin d'effectuer une séquence d'adresses dans l'ordre croissant ou décroissant, il suffit de choisir correctement les adresses visées de la mémoire principale. Comme nous venons de le voir dans le paragraphe traitant de la primitive d'écriture, il suffit d'incrémenter/décrémenter le registre contenant l'adresse.

Exemple d'implémentation du test MATS

Dans cette section nous présentons un exemple d'implémentation du test MATS sur un cache de données ayant 4 colonnes de 32 lignes.

Avant toutes choses, la mémoire principale doit être initialisée afin de contenir, avant le test, les mots de fond utilisés rangés aux adresses spécifiques. Le programme commence donc par une séquence d'instructions `sw` (store word), qui initialise 128 (4*32) mots consécutifs dans la mémoire principale, avec la valeur choisie. Cette séquence ne modifie pas le DCache puisque les données ne sont pas encore présentes. L'adresse de début de cette zone est rangée dans le registre `$1`. De la même façon une autre zone de la mémoire principale doit être initialisée avec les mots de fond inverses. L'adresse de début de cette zone est rangée dans le registre `$2`.

Le listing 4.1 présente l'implémentation de la primitive ⇑(w0). Cette séquence n'est pas sensible à la taille du cache. Quelle que soit la taille du cache, la taille du programme pour cette séquence sera toujours la même.

Listing 4.1: Implémentation de la primitive ⇑(w0)

```
1 elt0:
2  lw   $5, 0($1)     // w0 sur tout les bancs de la ligne (MISS)
3
4  // calcul de l'adresse suivante (-> up)
5  // $1 contient l'adresse de debut de la zone
6  // $3 contient l'adresse de fin de la zone
7  addi $1, $1, 16    // 16 dans le cas d'un cache ayant 4 bancs
8  bne  $1, $3, elt0  // tant que toutes les lignes n'ont pas ete ecrites
```

Le listing 4.2 implémente l'élément suivant de l'algorithme MATS. Le DCache est organisé physiquement en 4 bancs de mémoires RAM, le test de chaque banc n'a donc pas d'influence sur le test des autres. Ainsi, le test March de chaque banc doit être considéré de façon indépendante. Dans le listing 4.2, de la ligne 2 à 12 nous effectuons la lecture de chaque banc du cache. L'écriture ne doit venir

Listing 4.2: Implémentation de la primitive ⇑(r0,w1)

```
1  elt1:
2   lw   $5, 0($1)      // r0 banc 0: chargement du mot dans le registre $5
3   bne  $5, $10, fin   // $10 contient la valeur reference
4
5   lw   $5, 4($1)      // r0 banc 1 (HIT)
6   bne  $5, $10, fin   // r0 banc 1
7
8   lw   $5, 8($1)      // r0 banc 2 (HIT)
9   bne  $5, $10, fin   // r0 banc 2
10
11  lw   $5, 12($10)    // r0 banc 3 (HIT)
12  bne  $5, $10, fin   // r0 banc 3
13
14  // w1 sur tous les bancs (MISS)
15  // $2 contient la zone des mots de fond inverse
16  lw   $5, 0($2)
17
18  // calcul de l'adresse suivante (-> up)
19  // $3 contient l'adresse de fin de la zone
20  addi $1, $1, 16
21  addi $2, $2, 16
22  bne  $2, $3, elt1  // tant que toutes les lignes n'ont pas subi le test
```

qu'après la lecture de tous les bancs, puisque celle-ci affecte l'ensemble des bancs. Si l'on regarde les opérations sur les bancs de façon indépendante, chaque banc a subi une opération ⇑(r0,w1).

On peut distinguer l'influence de l'opération $r0$ sur la taille du code. En effet, cette opération doit être répétée pour chaque banc. Le nombre d'instructions dans cette séquence de code est proportionnel au nombre de colonnes du cache.

Le listing 4.3 présente le dernier élément du test MATS. Dans ce cas aussi, la lecture impose un code plus ou moins long suivant le nombre de colonnes du cache.

Nous avons vu dans cette section comment implémenter les primitives d'écriture et lecture March, en provoquant une succession de HIT et de MISS dans le cache. L'exemple de l'algorithme MATS nous a permis de voir comment construire des éléments March à partir de ces primitives. Par construction (par concaténation d'éléments March), n'importe quel algorithme March peut donc être implémenté pour le test du DCache.

4.2.3 Programme auto-testant le ICache

L'accès au ICache est plus restreint que celui du DCache, rendant plus difficile l'élaboration d'un programme de test. De plus, ce cache ne contient exclusivement que des instructions exécutées par le processeur. Ces restrictions s'incarnent sous la forme de trois contraintes : l'observabilité (lecture directe dans le cache impossible), le jeu d'instructions (contraintes sur les mots de fond), et la séquentialité (le test ne peux progresser que dans un seul sens).

Listing 4.3: Implémentation de la primitive ⇑(r1)

```
1  elt2:
2   lw   $5, 0($2)      // r1 banc 0: chargement du mot dans le registre $5
3   bne  $5, $11, fin   // $11 contient la valeur reference
4
5   lw   $5, 4($2)      // r1 banc 1 (HIT)
6   bne  $5, $11, fin   // r1 banc 1
7
8   lw   $5, 8($2)      // r1 banc 2 (HIT)
9   bne  $5, $11, fin   // r1 banc 2
10
11  lw   $5, 12($2)     // r1 banc 3 (HIT)
12  bne  $5, $11, fin   // r1 banc 3
13
14  // calcul de l'adresse suivante (-> up)
15  addi $1, $1, 16
16  addi $2, $2, 16
17  bne  $2, $3, elt2   // tant que toutes les lignes n'ont pas subi le test
```

Observabilité

Dans le DCache nous pouvons lire un mot et le comparer à celui attendu. Dans le ICache cette opération n'est pas possible. Le ICache pose un problème d'observabilité. Lorsque le microprocesseur lit une instruction dans le ICache, cette instruction est immédiatement exécutée. Par conséquent, le seul point d'observation possible est le résultat de l'exécution de l'instruction. Par exemple, le processeur exécute l'instruction de type I : `addi $1, $2, 5` (i.e. ajoute la valeur du registre `$2` avec la valeur 5, le résultat sera rangé dans `$1`). Si cette instruction est altérée lors de son passage dans le ICache, l'erreur peut se produire dans les différents champs de l'instruction (voir les formats du jeu d'instructions figure 3.3). Par exemple, dans cette instruction, la valeur immédiate (ici 5) peut être altérée. Si cette valeur est erronée, après l'exécution de l'instruction, le registre `$1` ne contiendra pas la valeur escomptée. Si l'erreur survient dans le codage du registre source `$2`, un autre registre sera utilisé. Dans le contexte de la faute unique, la valeur altérée du numéro de registre ne pourra prendre que les valeurs suivantes :`0,3,6,10,18`. Ainsi, si ces 5 registres sont initialisés avec des valeurs différentes de celle contenue dans `$2`, le résultat sera différent de celui attendu. Si l'erreur se produit dans le codage du registre destination `$1`, sa valeur restera inchangée. Pour finir, si l'erreur survient dans le champ "CODOP" une autre instruction sera exécutée. Donc, l'instruction doit être soigneusement choisie afin de pouvoir garantir que l'instruction altérée exécutée donnera un résultat différent de celui escompté.

Afin de garantir que l'instruction exécutée sera exactement celle attendue, nous devons mettre en place un contexte d'exécution garantissant l'observabilité. Ce qui signifie que l'instruction est exécutée dans un contexte spécifique, après exécution de cette instruction nous obtenons un contexte final. Si l'instruction exécutée diffère de celle rangée en mémoire principale (diffère d'un seul bit n'importe où dans l'instruction) l'exécution de cette instruction erronée conduira à un contexte final erroné. Le banc de registres peut être tout ou partie du contexte d'exécution. Ainsi, le contexte d'exécution permet de garantir l'exactitude de l'instruction exécutée.

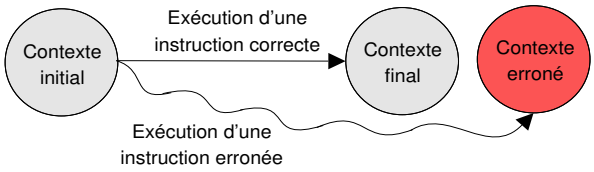


FIG. 4.4: L'exécution d'une instruction erronée conduit à un contexte d'exécution erroné, si un contexte initial a été correctement établi.

Le jeu d'instructions

Dans le ICache, les instructions sont les mots de fond utilisés. En fonction du nombre de CODOP disponibles (dans les architectures de type RISC ce nombre est limité par définition), la construction d'un mot de fond et de son inverse est en générale impossible. Ainsi, afin de réaliser un algorithme March, nous devons fractionner le mot de fond de 32 bits en mots de fond partiels. Nous devons créer autant de couples de mots de fond que nécessaire afin de couvrir l'ensemble des 32 bits de l'instruction. Par exemple, la figure 3.3 montre que si l'on utilise les instructions de type I, les 16 bits de poids faible sont aisément manipulables. Ainsi, nous pouvons facilement réaliser un couple de mots de fond partiels de 16 bits. Le test March doit être exécuté une fois avec ce premier couple de mots de fond partiels. Un ou plusieurs autres couples de mots de fond doivent être construits afin de tester les 16 bits de poids fort de l'instruction. Par exemple, nous pouvons réaliser un autre couple de mots de fond permettant de tester seulement 10 bits des 16 bits de poids fort (champs Rs/Rt). Ensuite, le test March doit être exécuté un seconde fois avec ce nouveau couple. Et ainsi de suite jusqu'à ce que les 32 bits de l'instruction soient couverts.

Séquentialité d'un programme

L'exécution d'un programme est séquentielle. Un bloc d'instructions est chargé dans le ICache, la première instruction est exécutée suivie de la seconde, puis de la troisième et ainsi de suite. Cette propriété séquentielle des programmes impose l'élément March suivant : $\Uparrow(\mathrm{w}a, \mathrm{r}a)$ où le mot de fond a est l'instruction à exécuter. En accord avec cette contrainte nous avons défini un algorithme March que nous avons appelé March IC : $\{\Uparrow(\mathrm{w}a, \mathrm{r}a)\,;\Uparrow(\mathrm{w}\bar{a}, \mathrm{r}\bar{a})\,;\Uparrow(\mathrm{w}a, \mathrm{r}a)\}$. Selon [Goor91], cet algorithme garantit la détection de toutes les fautes de collage et de toutes les fautes de transition.

Réalisation du programme de test

Un programme d'auto-test a été réalisé en respectant les trois contraintes que sont l'observabilité, le jeu d'instructions et la séquentialité. Les instructions de type I nous donnent le maximum de degré de liberté et un contexte d'exécution peut être facilement réalisé. L'exécution de ce type d'instructions est séquentielle. 26 bits peuvent être aisément manipulables (la valeur de l'immédiat, et les valeurs des registres source et destination). Un mot de fond partiel de 26 bits et de son inverse sont donc facilement définis :

```
DB    : xori $28, $28, 0x0000
invDB : andi $3 , $3 , 0xFFFF
```

Il ne reste plus que le test des 6 bits correspondants au "CODOP". La réalisation d'un deuxième mot de fond testant seulement 5 bits de ce champ est réalisable. En effet, aucune valeur du "CODOP" n'a exactement de valeur inverse. Un troisième couple de mots de fond doit donc être utilisé pour tester le dernier bit de ce champ.

Pour résumer, le test du ICache doit être exécuté en 3 passages avec l'algorithme March IC. Chaque passage utilisant un mot de fond partiel :

1. Test des 26 bits de poids faible, à l'aide d'instructions de format I comportant donc sur ce segment Rs, Rt et un immédiat sur 16 bits.
2. Test de 5 bits du "CODOP"
3. Test du dernier bit du "CODOP"

4.3 La plate-forme de simulation

La plate-forme de simulation (figure 4.5) utilisée est quasiment identique à celle précédemment utilisée pour le test du banc de registres (section 3.4). La plate-forme complète permet la co-simulation matérielle/logicielle. Les composants de la partie matérielle sont décrits en SystemC CABA (Cycle-Accurate, Bit-Accurate) provenant de la bibliothèque SOCLIB [SoCLIB]. Le processeur utilisé est le MIPS R3000 à 5 étages, connecté à un cache dont les caractéristiques sont détaillées plus bas. Finalement, une mémoire principale, contenant le programme de test, est connectée à l'interconnect système (cross-bar respectant la norme VCI [VSIA]).

4.3.1 Déformabilité du cache

L'architecture du cache utilisé est celle décrite à la section 4.2.1. La taille du cache peut varier en fonction des paramètres L (nombre de lignes) et C (nombre de colonnes). Les valeurs L et C peuvent être différentes pour le cache instructions et le cache de données. Le nombre de lignes L peut prendre les valeurs 64, 128, 512 et 1024. Le nombre de bancs mémoire ou colonnes C peut prendre les valeurs 2, 4, 8 et 16. Le tableau 4.2 présente les différentes tailles du cache obtenues selon ces deux paramètres. Chaque cache (instructions et données) peut donc avoir une taille qui varie entre 512 octets et 32Ko.

	# Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	0.5 Ko	1 Ko	2 Ko	4 Ko
128	1 Ko	2 Ko	4 Ko	8 Ko
256	2 Ko	4 Ko	8 Ko	16 Ko
512	4 Ko	8 Ko	16 Ko	32 Ko

TAB. 4.2: Taille du cache en fonction des paramètres L et C.

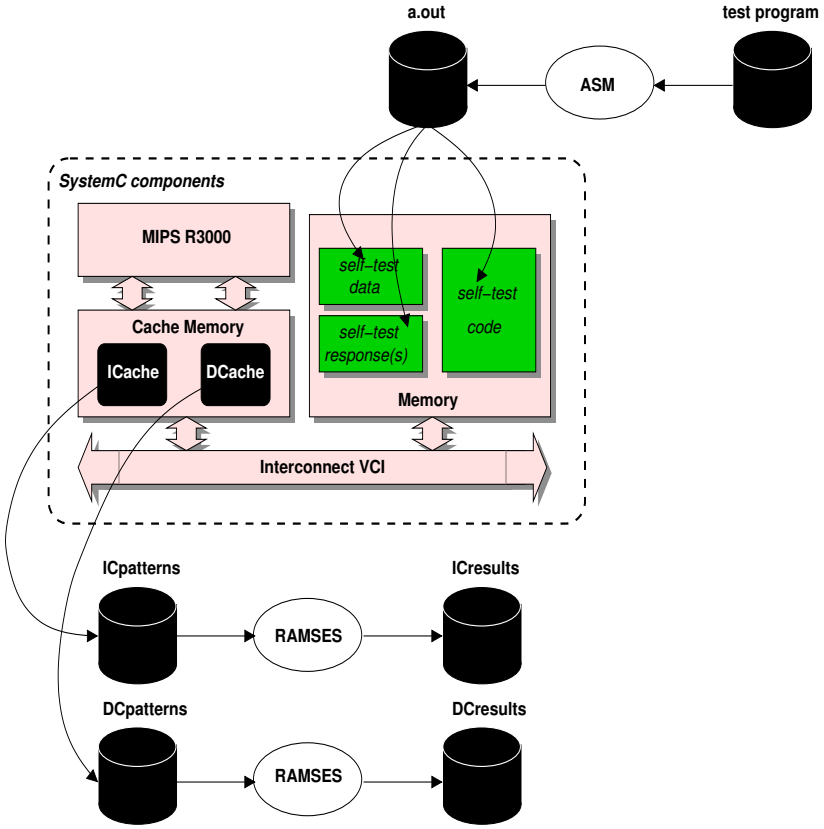


FIG. 4.5: Plate-forme de test des mémoires caches

4.3.2 Génération automatique des programmes de test du DCache

Dû à l'approche systématique de la génération des tests March pour le DCache, un outil a été développé afin de générer automatiquement les programmes d'auto-test. Cet outil prend en entrée un fichier décrivant le test March à implémenter ainsi que les mots de fond utilisés. Ce fichier d'entrée est le même que celui utilisé pour la génération des programmes de test du banc de registres (voir section 3.4.1). A la différence du générateur de programmes de test pour banc de registres, cet outil prend comme paramètre d'entrée la taille du cache (nombre de lignes, nombre de colonnes).

4.3.3 RAMSES et fichiers de traces

Pendant l'exécution, les mémoires testées (ICache et DCache) génèrent des fichiers de traces. Pour être plus précis, il existe un fichier de traces propre à chaque banc mémoire (colonne) de chacun des caches. Ces traces sont envoyées au logiciel RAMSES, permettant de vérifier les taux de couverture obtenus. Les fichiers de traces générés ainsi que le logiciel RAMSES sont décrits dans la section 3.4.2.

4.4 Résultats expérimentaux pour le DCache

Comme pour le banc de registres, les cinq algorithmes MATS, MATS+, MATS++, MARCH X et MARCH C- présentés dans le tableau 2.1 ont été utilisés pour effectuer le test du DCache. De la même façon, pour chaque test, deux versions ont été réalisées. La première version n'utilisant qu'un mot de fond (version 1 DB). La deuxième version utilisant les six mots de fond (version 6 DBs). Ainsi, les simulations des 5 tests March (en deux versions) ont été effectuées sur les 16 configurations possibles du DCache.

4.4.1 Taux de couverture

Le tableau 4.3 présente les taux de couverture calculés par RAMSES. Ce tableau nous permet de voir la qualité du test qui peut être obtenu sur le DCache. Ces résultats correspondent aux valeurs théoriques attendues (cf tableau 2.2).

Test	DB	SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
MATS	1	100.0 %	50.0 %	49.2 %	74.6 %	50.0 %	25.0 %
	1–6	100.0 %	100.0 %	98.5 %	98.5 %	99.2 %	88.3 %
MATS+	1	100.0 %	50.0 %	99.2 %	50.0 %	74.8 %	37.4 %
	1–6	100.0 %	100.0 %	100.0 %	98.4 %	99.2 %	91.0 %
MATS++	1	100.0 %	100.0 %	99.2 %	50.0 %	75.2 %	37.6 %
	1–6	100.0 %	100.0 %	100.0 %	98.4 %	99.2 %	91.1 %
MARCH X	1	100.0 %	100.0 %	99.2 %	62.4 %	100.0 %	50.0 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	100.0 %	99.1 %
MARCH C-	1	100.0 %	100.0 %	99.2 %	99.6 %	100.0 %	99.6 %
	1–6	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %

TAB. 4.3: Taux de couverture obtenus après simulation par les programmes auto-testants.

4.4.2 Temps de test

Il n'est pas question ici de reporter les temps de test pour toutes les simulations effectuées. Une sélection des temps de test a été effectuée pour certaines tailles du cache et sont reportées dans le tableau 4.4. Cinq tailles ont été sélectionnées. La première et la dernière colonne désignent respectivement la plus petite (512 octets) et la plus grande taille (32 Ko) de cache utilisée. Des tailles intermédiaires ont été sélectionnées (ces tailles sont en général celles utilisées dans les SoCs), 2Ko, 4Ko et 8Ko.

Le tableau 4.4 nous permet de distinguer les deux valeurs extrêmes des temps de test. A une extrémité se trouve le test MATS version 1 DB pour un cache de 512 octets. Environ 5000 cycles suffisent à appliquer ce test. A l'autre extrémité, le test MARCH C-, en version 6 DBs, pour un cache de données de 32Ko nécessite l'application d'un peu moins de deux millions de cycles. Pour les caches de données de 2Ko, 4Ko et 8Ko les temps de test sont de l'ordre de la dizaine de millier de cycles en moyenne pour les March en version 1 DB, de l'ordre de la centaine de milliers de cycles pour les versions 6 DBs.

Test	DB	*0.5 Ko* $64L * 2C$	*2 Ko* $64L * 8C$	*4 Ko* $128L * 8C$	*8 Ko* $256L * 8C$	*32 Ko* $512L * 16C$
MATS	1	5	13	25	50	180
	1–6	30	75	150	300	1100
MATS+	1	6	13	27	55	200
	1–6	35	81	160	320	1160
MATS++	1	6	16	31	62	227
	1–6	36	94	185	370	1360
MARCH X	1	7	16	32	65	230
	1–6	38	96	190	380	1400
MARCH C-	1	10	23	46	90	325
	1–6	60	140	274	550	1950

TAB. 4.4: Nombre de cycles pour les tests March pour différentes tailles du DCache (en milliers de cycles).

La figure 4.6 représente graphiquement les temps de cycles pour un DCache de 4Ko. La première information qui ressort de cette figure, est que les versions 6 DBs nécessite environ six fois plus de temps que les versions 1 DB. La deuxième, est que, pour une taille du cache donnée, tous les tests March sont dans la même fourchette de temps de test. En effet entre le test le plus long, MARCH C-, et le plus court, MATS, le rapport n'est que de **1.8**.

Il est intéressant de noter que le test MARCH C- en version 1 DB, bien que 3.24 fois moins long à exécuter que le test MATS en version 6 DB, permet d'aboutir à une meilleure qualité de test (cf tableau 4.3).

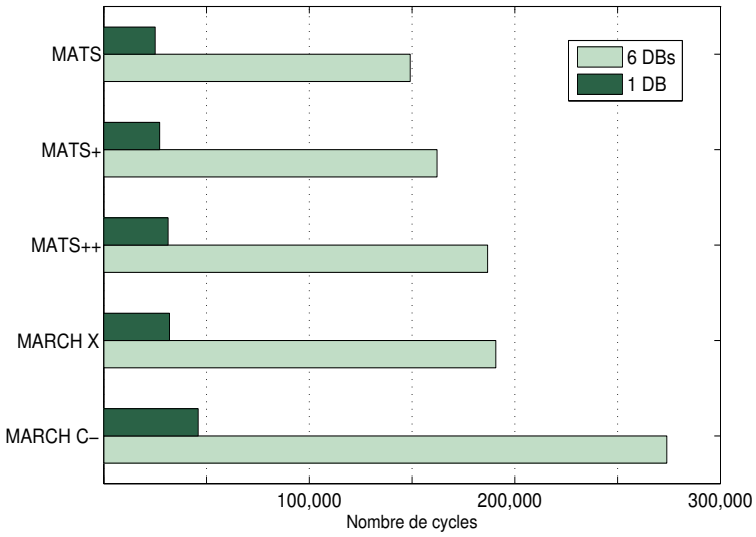


FIG. 4.6: Nombre de cycles pour les tests March pour un cache de données de 4Ko.

Influence de la configuration du cache

Afin d'examiner l'impact de la configuration du DCache sur les temps de test, le tableau 4.5 présente le nombre de cycles nécessaires à l'exécution du test MATS++ (version 6 DBs) pour toutes les configurations du DCache. Pour rappel, la taille du cache varie de 512 octets ($64L$*$2C$) à 32 Ko ($512L$*$16C$).

	# Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	36,509	55,472	93,780	**170,564**
128	71,892	110,064	**186,772**	340,356
256	142,676	**219,248**	372,756	679,940
512	**284,244**	437,616	744,724	1,359,108

TAB. 4.5: Nombre de cycles nécessaires au test MATS++ pour différentes configurations du DCache.

Le tableau met en évidence l'influence de la configuration du cache sur le temps de test. En effet, la diagonale mise en caractère gras représente les temps de test pour une même taille du cache. Les configurations $512L * 2C$, $256L * 4C$, $128L * 8C$ et $64L * 16C$ représentent un cache de donnée de 4Ko. Or, bien que les caches soient de la même taille, les temps de test correspondant varient de 170 000 à 280 000 cycles (rapport de **1.66**). En fait, ce phénomène est tout simplement dû à la capacité du DCache à ramener une ligne entière (x colonnes) depuis la mémoire principale à chaque lecture effectuant un "MISS" de cache. Plus la ligne est grande, moins les accès à la mémoire principale sont nombreux.

Comparaison avec une approche BIST

Puisque nous ne pouvons trouver dans la littérature SBST des temps de test sur des mémoire caches, nous allons nous comparer avec une approche classique de type BIST.

Nous avons opté pour une stratégie dont le contrôleur BIST est partagé entre chaque banc mémoire du cache. Un BIST dédié à chaque banc mémoire du cache aurait engendré une surface de test trop importante. En effet, pour un cache de 2 Ko composé de 4 bancs de 512 octets, la surface dédiée au BIST pour chaque banc est d'environ 30% (voir figure 1.6 du chapitre Problématique). Etant donnée la proximité physique des bancs mémoire dans le cache, le contrôleur BIST peut envoyer les bits de données et d'adresses en parallèle. En d'autres termes, le contrôleur BIST peut effectuer une opération de lecture ou d'écriture en un cycle sur chaque banc. Le test des bancs est effectué un par un. Ainsi, le nombre de cycles nécessaires pour effectuer un test March avec cette approche BIST est donc :

$$NC_{BIST} = L * N_{MarchPrim} * N_{DB} * C \tag{4.1}$$

Où L et C représentent la configuration du DCache, $N_{MarchPrim}$ est le nombre de primitives March dans l'algorithme et N_{DB} est le nombre de mots de fond sur lesquels l'algorithme doit être itéré.

Le tableau 4.6 donne le nombre de cycles nécessaires à l'approche BIST imaginée pour effectuer le test du cache. Ces chiffres sont donnés pour le test MATS++ en version 6 DBs. On peut voir que, contrairement à l'approche SBST, l'approche BIST n'est absolument pas sensible à la configuration du cache. Ces résultats peuvent être mis en parallèle avec les résultats obtenus par l'approche SBST reporté dans le tableau 4.5.

Le tableau 4.7 présente le rapport des temps entre l'approche SBST et l'approche BIST. Le tableau 4.7 permet de mettre en évidence l'influence du nombre de colonnes sur ce rapport. Pour une configuration d'uniquement 2 bancs, l'approche BIST est environ 7.8 fois plus rapide que l'approche SBST. Ce rapport descend à 4.6 pour une configuration à 16 bancs (pour un test MATS++ version 6 DBs). Quelle que soit la taille du cache, seul le nombre de bancs mémoire influence ce rapport. Ceci est dû au fait que l'approche SBST est sensible au nombre de bancs mémoire présents dans le cache au contraire de l'approche BIST. La figure 4.7 présente de façon graphique cette évolution pour chaque test March.

	# Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	4608	9216	18432	36864
128	9216	18432	36864	73728
256	18432	36864	73728	147456
512	36864	73728	147456	294912

TAB. 4.6: Nombre de cycles nécessaires au test MATS++ (6 DBs) pour différentes configurations du DCache pour une approche BIST.

	# Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	7.92	6.02	5.09	4.63
128	7.8	5.97	5.07	4.62
256	7.74	5.95	5.06	4.61
512	7.71	5.94	5.05	4.61

TAB. 4.7: Rapport SBST/BIST pour le test MATS++ pour différentes configurations du DCache.

4.4.3 Taille des programmes de test

Après avoir vu les taux de couverture obtenus par les programmes de test ainsi que le nombre de cycles nécessaires à leur application, examinons leur taille. Ce facteur est important puisqu'il impacte le temps de test global. Comme nous l'avons précisé, le programme est chargé par l'ATE à la fréquence de celui-ci. Ainsi, plus le programme est léger, moins le temps de test est long.

Le tableau 4.8 reporte les tailles des différents programmes implémentant les tests March étudiés. Ce tableau ne présente que le nombre de colonnes du DCache, puisque le nombre de lignes n'a

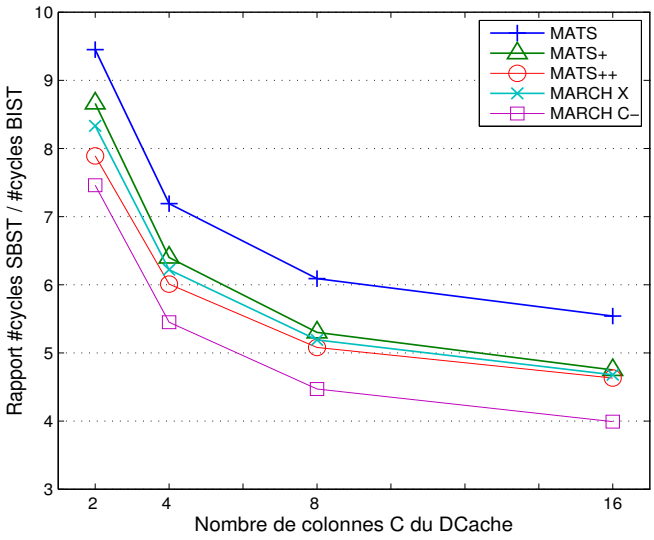


FIG. 4.7: Rapport SBST/BIST en fonction du nombre de colonnes C dans le DCache pour chaque test March.

aucun impact sur la taille des programmes générés (voir section 4.2.2). Ainsi, un programme de test implémentant l'algorithme March X (version 6 DBs) pour une configuration $64L*8C$ (soit un DCache de 2Ko) ou une configuration de $512L * 8C$ (16Ko) ont la même taille : 812 octets.

Les programmes générés sont très légers, inférieurs à 1Ko. La moyenne pour les versions 1 DB est de **471** octets. Elle est de **788** octets pour les versions 6 DBs (soit 1.7 fois plus gros que les versions 1 DB). Nous ne pouvons pas comparer la taille des programmes à d'autre approches SBST sur les mémoires caches (faute de résultats). Néanmoins nous pouvons rappeler, afin de donner un ordre de grandeur, que l'approche proposée dans [Gizopoulos04] pour le test d'un banc de registres de 128 octets utilise un programme de test de 2.8Ko.

Test	DB	*2*	*4*	*8*	*16*
MATS	1	280	312	376	504
	1–6	584	616	680	808
MATS+	1	284	316	380	508
	1–6	588	620	684	812
MATS++	1	300	348	444	636
	1–6	604	652	748	940
MARCH X	1	348	396	492	684
	1–6	668	716	812	1004
MARCH C-	1	484	564	724	1044
	1–6	836	916	1076	1396

TAB. 4.8: Taille des programmes de test en octets en fonction du nombre de colonnes.

4.5 Résultats expérimentaux pour le ICache

Les programmes de test pour le DCache fournissent un test de qualité, sont légers et s'exécutent en 6 fois plus de temps qu'une approche BIST partagée. Voyons ce qu'il en est des programmes de test du ICache.

Pour rappel, les programmes de test du ICache implémentent un nouvel algorithme March appelé March IC $\{\Uparrow(\mathrm{w}a, \mathrm{r}a)\,;\Uparrow(\mathrm{w}\bar{a}, \mathrm{r}\bar{a})\,;\Uparrow(\mathrm{w}a, \mathrm{r}a)\}$, respectant les contraintes liées au fait que ce cache ne contienne que des instructions valides. Ainsi, contrairement aux programmes d'auto-test du DCache qui peuvent être générés de façon automatique, ceux du ICache doivent être créés à la main et nécessite un certain degré de connaissance du processeur. Dans le cas du MIPS, le test March IC doit être répété sur 3 mots de fond partiels.

4.5.1 Taux de couverture

Les taux de couverture fournis par RAMSES après exécution du programme de test sur le ICache sont reportés dans le tableau C.3. Comme prévu, toutes les fautes de collage (SAF) et de transition (TF) sont détectées. Cependant, RAMSES révèle que l'impact du test March IC sur les autres modèles de fautes est bas. Effectivement, moins de 3% des fautes d'adresses (AF) sont détectées, comme 2% et 3% des fautes de couplage CF_{id} et CF_{in}. On peut noter tout de même que les fautes de couplage d'état (CF_{st}) atteignent presque 80%.

SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
100.0 %	100.0 %	2.7 %	78.7 %	3.0 %	2.0 %

TAB. 4.9: Taux de couverture du test March IC.

4.5.2 Temps de test

Le tableau C.2 contient le nombre de cycles nécessaires à l'application du test March IC pour différentes configurations du ICache. Pour un cache de 512 octets ($64L$*$2C$), 7513 cycles sont suffisants. Pour un cache de 32Ko ($512L$*$16C$), 186,713 cycles sont nécessaires.

	#Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	7,513	9,803	14,328	23,545
128	14,873	19,467	28,600	46,721
256	29,593	38,795	57,144	93,385
512	59,033	77,451	114,232	186,713

TAB. 4.10: Nombre de cycles nécessaires au test March IC pour différentes configurations du ICache.

Comparaison avec une approche BIST

Nous pouvons comparer le March IC en version logicielle (SBST) avec le March IC en version matérielle (BIST). Les résultats en nombre de cycles pour cette dernière version sont obtenus en utilisant la formule 4.1 avec les attributs du March IC. La figure 4.8 présente le rapport entre les temps de test du March IC version SBST et les temps de test du March IC version BIST. Comme pour le DCache, seul le nombre de colonnes influence ce rapport.

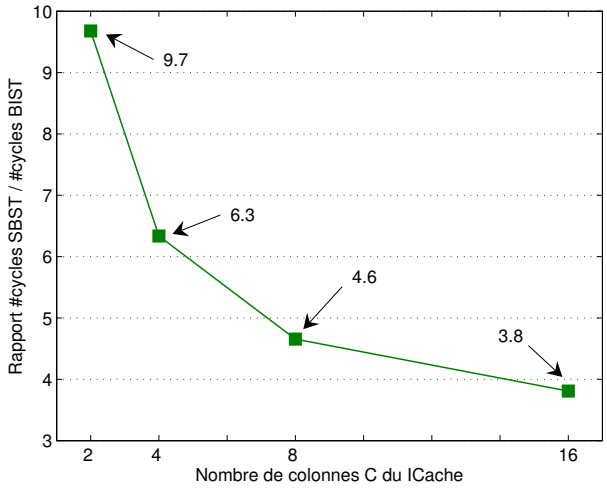


FIG. 4.8: Rapport des temps de test SBST/BIST en fonction du nombre de colonnes C dans le ICache.

4.5.3 Taille des programmes de test

Un des points faibles de cette méthode réside dans la taille des programmes du ICache. Le tableau 4.11 répertorie ces résultats. Les détails de la configuration du ICache ne sont pas présentés puisque seule la taille globale du ICache impacte la taille des programmes de test. Les résultats montrent que les programmes sont **9** fois plus gros que le cache testé. Ainsi, pour un cache de 4Ko, un programme de 36Ko est nécessaire. Pour un cache de 32Ko, pas moins de 288Ko de programme de test sont nécessaires. Ces résultats étaient attendus. En effet, l'execution d'un élément March de type $\Uparrow$(wa, ra) nécessite la présence en mémoire d'une séquence d'instructions de la taille du cache. Or le March IC répète cet élément March 3 fois, et l'implémentation sur le MIPS a nécessité la répétition du March IC sur 3 mots de fond partiels. A ce stade nous devons rappeler que dans les méthodes SBST le programme de test est chargé par l'équipement de test bas coût fonctionnant à basse fréquence. Cette phase peut dominer le temps de test total.

ICache (en Ko)	*1*	*2*	*4*	*8*	*16*	*32*
Programme (en Ko)	9	18	36	72	144	288

TAB. 4.11: Tailles des programmes de test en fonction de la taille du ICache.

4.6 Conclusion

Lorsque les processeurs embarqués exécutent des programmes rangés dans la mémoire principale du SoC, ces programmes sont chargés par fragments dans leurs mémoires caches. Ainsi, bien organisés en mémoire principale, ces programmes peuvent, par effet de bord, tester les mémoires caches du processeur. Nous avons pu voir dans ce chapitre comment implémenter des programmes de test permettant d'exécuter des tests March, par effet de bord, sur les caches instructions et les caches de données.

Le cache de données

La facilité d'accès à ce cache permet l'implémentation de n'importe quel test March. De plus, de part leur régularité, la génération des programmes de test peut être automatisée. La qualité du test dépend du test March effectué et peut conduire à une couverture de fautes élevée (ex : MARCH C-). Les temps de test pour des caches de données de 2Ko à 8Ko sont de l'ordre de la centaine de milliers de cycles. Ces résultats ont été comparés à une approche BIST partagée. Une approche BIST se révèle être 4 à 9 fois plus rapide (6 fois en moyenne), mais se fait au détriment de la surface supplémentaire qui est nulle dans le cas du SBST. Finalement les programmes de tests sont très légers, de l'ordre de 600 octets en moyenne, ce qui est un point essentiel des stratégies SBST.

Le cache instructions

Le cache instructions impose certaines restrictions ne nous permettant pas d'implémenter n'importe quel test. Un test March spécifique a été développé : le March IC. Ce test ne garantit la détection totale que des fautes de collage (SAF) et de transition (TF). De plus, l'élaboration de ce programme nécessite une connaissance détaillée du jeu d'instructions employé. Les temps de test comparés à une approche BIST sont en moyenne 6 fois plus élevés pour le même type de test. Un défaut de cette méthode réside dans la taille des programmes de test. En effet, les programmes font 9 fois la taille du cache instructions testé. Un cache instructions de 2Ko nécessite un programme de test de 18Ko.

Le tableau 4.12 résume les différentes caractéristiques du test du cache de données et du cache instructions.

	Cache de données	*Cache instructions*
Génération	Automatique	Manuelle
Couverture	Fonction du March implémenté	SAF et TF garantie
Temps de test Min. *Temps de test Max.* *Temps de test Moy.*	$\approx$ 5 000 cycles $\approx$ 2 000 000 cycles $\approx$ 130 000 cycles	$\approx$ 7 000 cycles $\approx$ 200 000 cycles $\approx$ 50 000 cycles
Temps de test moyen par rapport à un BIST	6 fois plus long	6 fois plus long
Taille des programmes	$\approx$ 600 octets	9 * taille du ICache (ex : ICache=4Ko $\rightarrow$ 36Ko)

TAB. 4.12: Caractéristiques du test logiciel des mémoires caches

Chapitre 5

Conclusion

Le BIST matériel pour les mémoires est une solution de test efficace. En effet, l'ajout de la circuiterie de test est aujourd'hui automatisé, les algorithmes de test peuvent être choisis en fonction de la qualité du test désirée, et le temps d'application est généralement court.

Malheureusement, la surface engendrée par la circuiterie BIST pour des petites mémoires de l'ordre du kilo-octets peut s'avérer excessive. Or, les processeurs embarqués dans les SoCs sont truffés de petites mémoires, telles que les bancs de registres (centaines d'octets), les caches instructions et les caches de données (de l'ordre du kilo-octets). Ajouter du matériel BIST à toutes ces mémoires engendrerait une surface dédiée au test trop importante.

D'un autre côté, une alternative à l'auto-test matériel (BIST) a été présentée : l'auto-test logiciel (SBST). Le SBST, de part son aspect logiciel, a la particularité de n'ajouter aucune surface supplémentaire.

Le choix d'une stratégie de test est en général basé sur un compromis entre plusieurs facteurs tels que le temps de test, la surface supplémentaire et la qualité du test. Ainsi, dans quelle mesure vaut-il mieux adopter une stratégie logicielle plutôt que matérielle ?

C'est pourquoi, dans cette première partie du manuscrit nous nous sommes intéressés à l'auto-test logiciel des mémoires embarquées dans les processeurs. Nous avons proposé des stratégies de test pour les bancs de registres et les mémoires caches. Les résultats obtenus nous permettent d'ébaucher des conclusions sur le choix d'une politique de test logiciel ou matériel.

Nous pouvons évaluer l'efficacité du SBST sur les trois composants (banc de registres, cache de données et cache instructions) selon les quatre critères suivants :

- **Génération du programme de test.** Elle peut être automatisée ou manuelle.
- **Qualité du test.** Quelles sont les couvertures de test accessibles ?
- **Temps de test.** Il s'agit du nombre de cycles nécessaires à l'exécution du programme.
- **Taille des programmes de test.** Le temps de chargement du programme dans le SoC influe sur le temps de test global. Plus le programme est petit, plus le temps de test sera court.

Le banc de registres

Nous avons présenté une méthode systématique pour la génération des algorithmes March, basée sur la tripartition du banc de registres. Les résultats obtenus nous permettent d'évaluer la méthode selon les quatre critères présentés précédemment :

- **Génération du programme de test :** automatique.
- **Qualité du test :** celle désirée, fonction du test March employé.
- **Temps de test :** de l'ordre du millier de cycles. Comparé à l'approche [Gizopoulos04], et seules les SAFs sont considérées, les temps de test sont équivalents.
- **Taille des programmes de test :** de 2.5 Ko à 7.5 Ko. Comparé à l'approche [Gizopoulos04], et seules les SAFs sont considérées, la taille des programmes de test est équivalente.

La génération d'un programme de test pour ce composant est automatisée et la qualité est fonction de celle désirée. Sachant qu'un BIST dédié au test d'un banc de registres de 1 kilo-bits occupe 60% de la surface totale [Nadeau-Dostie90], ceci penche en faveur de l'emploi du SBST pour ce composant.

Les mémoires caches

Nous avons proposé une stratégie de test par effets de bord. Notre étude a porté sur des mémoires caches à correspondance directe et dont la politique d'écriture en mémoire est "Write-Through". Les différences de fonctionnement entre le cache de données et le cache instructions ont conduit à des stratégies de test ainsi qu'a des résultats différents.

Le cache de données

La facilité de gestion du contenu dynamique de ce cache, en provoquant une série de MISS et de HIT, nous permet d'implémenter n'importe quel test March. Les résultats obtenus nous permettent d'évaluer cette méthode d'auto-test logiciel :

- **Génération du programme de test :** automatique.
- **Qualité du test :** celle désirée, fonction du test March employé.
- **Temps de test :** de l'ordre de la centaine de milliers de cycles. Comparé à une approche BIST, les temps de test sont en moyenne 6 fois plus longs, à qualité de test équivalente.
- **Taille des programmes de test :** de l'ordre de 500 octets. Très légers.

Le choix entre une stratégie BIST ou SBST pour ce cache est un compromis entre la surface et le temps de test. On préférera le BIST si le temps de test est un facteur décisif et qu'en même temps la surface supplémentaire est secondaire. Dans les autres cas le SBST parait être une alternative très intéressante.

Le cache instructions

Le cache instructions impose certaines restrictions ne nous permettant pas d'implémenter n'importe quel test. En effet, il doit satisfaire aux contraintes d'observabilité, du jeux d'instructions et de la séquentialité des programmes. Un test March spécifique a été développé : le March IC. Regardons les résultats obtenus selon les quatre critères :

- **Génération du programme de test :** manuelle.
- **Qualité du test :** SAF, et TF garantis.
- **Temps de test :** de l'ordre de 50 000 cycles. Comparé à une approche BIST, les temps de test sont en moyenne 6 fois plus longs, à qualité de test équivalente.
- **Taille des programmes de test :** de l'ordre de 10 à 100 Ko. Très gros.

Le choix entre une stratégie BIST ou SBST pour ce cache est un compromis entre la surface supplémentaire, le temps de test et la qualité du test. Le SBST pour ce cache peut être une alternative intéressante dans le cas ou le cache est relativement petit (environ 1 Ko), et la qualité de test désirée est moyenne. Evidemment, si la surface supplémentaire est un critère rédhibitoire, la stratégie logicielle reste l'approche de choix.

Deuxième partie

Auto-test logiciel des SoCs : utilisation d'un micro-testeur embarqué

Chapitre 6

Introduction

Comme nous l'avons vu, l'auto-test logiciel (SBST) peut être divisé en deux grandes branches. La première se focalise sur l'auto-test logiciel des processeurs (embarqués ou hautes performances). La deuxième branche, l'auto-test logiciel des SoCs, propose de tester les composants du système grâce au processeur embarqué.

Dans la première partie de ce manuscrit, nous avons abordé l'auto-test logiciel des processeurs. Plus particulièrement, nous avons proposé des méthodes de développement spécifiques aux mémoires embarquées dans les processeurs.

Dans cette deuxième partie nous aborderons l'auto-test logiciel des SoCs. Nous allons présenter une méthode de test générique s'intégrant dans le flot de conception de test classique, en proposant une approche compatible avec la norme IEEE 1500.

L'auto-test logiciel des SoCs fait l'hypothèse qu'au moins un coeur de processeur est embarqué dans le système. Ainsi, il peut être réutilisé pour tester les autres composants du système. Il est utilisé comme générateur de vecteurs de test, appliquant ces vecteurs, à travers l'interconnect système, aux coeurs à tester. Les réponses sont ensuite récupérées par le processeur et rangées en mémoire. Cette approche aborde le test de la puce d'un point de vue *fonctionnel*.

D'un autre côté, d'un point de vue plus *structurel*, la norme IEEE 1500 se propose de standardiser l'accès aux fonctionnalités test des coeurs. En effet, elle propose l'élaboration d'un anneau d'isolation (wrapper) de test autour de chaque coeur. Ainsi, chaque coeur est fourni avec son wrapper de test ainsi que ses vecteurs de test. L'intégrateur système doit alors concevoir un mécanisme d'accès (TAM) à ces wrappers de test, lui permettant de délivrer les vecteurs de test et de récupérer les réponses des coeurs.

Comment concilier les avantages de l'auto-test logiciel avec les avantages de la compatibilité IEEE 1500 ?

Cette partie est divisée en 5 chapitres.

Le chapitre suivant présente l'état de l'art. Nous allons étudier les différentes stratégies d'auto-test logiciel des SoCs. L'étude des avantages et des inconvénients de ces différentes stratégies nous permettra de concevoir une approche d'auto-test logiciel compatible avec la norme IEEE 1500.

Le chapitre 8 présentera la stratégie adoptée. Nous verrons quels sont les composants matériels nécessaires à la mise en place de notre stratégie. Dans ce cadre, nous exposerons le déroulement du processus de test global.

L'implantation de la stratégie proposée n'est pas sans conséquences sur la conception des wrappers de test et des coeurs. Le chapitre 9 traitera des implications de notre stratégie, d'un point de vue matériel et temporel. Ce chapitre sera l'occasion de proposer une implémentation du test *at-speed* des coeurs compatibles IEEE 1500.

Le chapitre 10 exposera les plates-formes de simulations utilisées. Les résultats obtenus par notre stratégie seront examinés sous différents aspects, tel que le temps de test, le volume des données de test et la surface supplémentaire. Enfin, les temps de test de notre approche seront comparés à ceux d'un TAM classique, piloté par un testeur externe, utilisé dans l'industrie. Cette comparaison nous permettra de savoir si la stratégie proposée est viable en terme de temps de test.

Finalement le chapitre 11 conclura cette deuxième partie en récapitulant les points essentiels de notre stratégie.

Chapitre 7

État de l'art

Dans ce chapitre, nous allons présenter l'état de l'art de l'auto-test logiciel des SoCs. Nous verrons quelles sont les stratégies mises en oeuvre permettant le test des composants d'un SoC.

Ainsi, de la littérature scientifique publiée sur le sujet, on peut distinguer trois types de stratégies permettant d'effectuer le test logiciel des différents coeurs d'un SoC :

- La première approche purement fonctionnelle, n'a aucun recours à la DfT.
- La deuxième approche propose l'utilisation de wrappers de test.
- La troisième approche préconise l'utilisation de coeurs dédiés au test de la puce : les micro-testeurs embarqués.

Nous mettrons en exergue, les avantages et les inconvénients associés à chaque approche.

La conclusion de ce chapitre récapitulera les caractéristiques de chaque approche, afin d'établir une stratégie de test palliant les défauts des approches proposées.

7.1 Approches purement fonctionnelles

Commençons notre étude par les approches purement fonctionnelles proposées dans la littérature. Ces approches ne font aucun recours à la DfT. Ce qui signifie que l'ajout de surface supplémentaire dédié au test est nulle. Ces types d'approches utilisent le processeur embarqué pour générer les vecteurs de test (réutilisé comme *source*). L'interconnect système sert à véhiculer ces vecteurs jusqu'au coeur testé (réutilisé comme *TAM*). Enfin, le microprocesseur récupère les réponses et les analyse (réutilisé comme *sink*). La figure 7.1 expose ce concept.

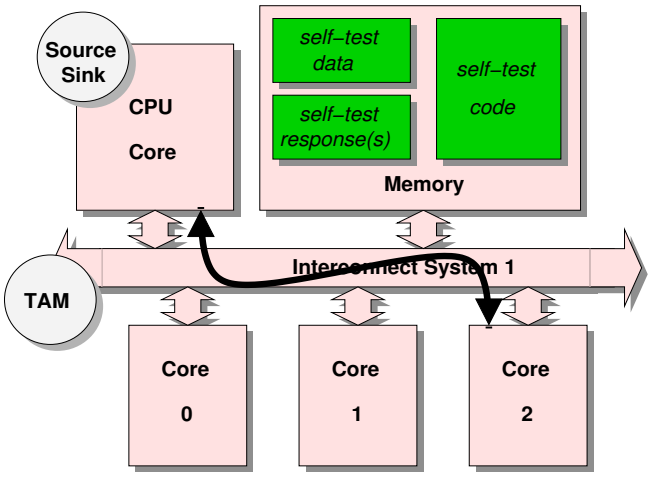


FIG. 7.1: Approches purement fonctionnelles

7.1.1 Méthodes fonctionnelles spécifiques à chaque coeur

Jacob Abraham et al. dans l'article [Jayaraman02] proposent d'utiliser ce type d'approche purement fonctionnelle pour le test d'un SoC. Ce système contient, autour d'un bus à base de tri-state, un processeur (Intel 8085), un contrôleur DMA, un contrôleur de mémoire externe, un port parallèle, un port série (USART) et un contrôleur de bus. Le processeur Intel 8085 est utilisé pour générer les vecteurs de test à appliquer aux coeurs du système et pour analyser les réponses données par ces derniers. Les résultats sur le test de l'Intel 8251 Universal Synchronous Asynchronous Receiver Transmitter (USART) sont présentés. L'USART est un petit coeur d'environ 3000 portes logiques et 270 bascules. Le taux de couverture obtenu est de moins de 68%. D'après les auteurs, la raison principale d'un taux de couverture si faible est dû à l'incapacité à observer beaucoup de fautes.

Les problèmes liés à cette approche sont :
- La nécessité de connaître la fonctionnalité de tous les coeurs testés puisque la génération du programme de test est intiment couplée aux fonctionnalités du composant. Ceci rend difficile la génération automatique du test.
- Le coeur étant immergé dans le système, le programme de test du coeur est aussi conçu en fonction du système environnant. C'est pourquoi le programme de test doit être révisé à chaque intégration dans un nouveau système.

– La faible contrôlabilité et observabilité d'un coeur dans son environnement fonctionnel ne permet pas d'obtenir des taux de couverture satisfaisant.
– L'interconnect est utilisé pour véhiculer les vecteurs de test, donc, une connexion entre les coeurs testés et l'interconnect est obligatoire.

7.1.2 Méthodes fonctionnelles génériques

Les auteurs de l'article [Tehranipour03] proposent une méthode générique pour le test des composants d'un SoC. Chaque composant possède un programme de test stocké dans une mémoire externe. Chaque programme est ramené à l'intérieur du SoC par un système de DMA. Le programme de test exécuté par le microprocesseur embarqué comporte 4 étapes : i) la procédure de génération des vecteurs, ii) la procédure de distribution des vecteurs, iii) la procédure de génération de la signature et iv) la procédure de comparaison. Lors du test des coeurs, la procédure de génération des vecteurs procède de façon pseudo-aléatoire conduisant à un taux de couverture assez faible (environ 84%).

Les problèmes liés à cette approche sont :
– La génération de vecteurs pseudo-aléatoire ne permet pas d'obtenir des taux de couverture satisfaisant.
– Le taux de couverture peu élevé est aussi dû à la faible contrôlabilité et observabilité du coeur dans son environnement fonctionnel.
– Les auteurs font l'hypothèse, comme dans la méthode précédente, que les coeurs testés sont connectés à l'interconnect système.

7.2 Approches avec wrapper de test

De la même façon que les approches purement fonctionnelles, ces techniques utilisent l'interconnect système comme TAM et le processeur embarqué comme *test source* et *test sink*. Afin de combler le manque de contrôlabilité et d'observabilité du coeur sous test, des chercheurs ont proposé l'utilisation d'un wrapper de test autour du coeur testé. Tout d'abord nous verrons l'approche proposée dans [Papachristou99], où les wrappers permettent de s'affranchir de la connexion avec l'interconnect système. Le wrapper, dans [Hwang01], permet le contrôle et l'observation des entrées/sorties de chaque coeur. Finalement, dans [Huang01, Iyer02], le wrapper permet la réutilisation des structures DfT fournies par le coeur.

L'introduction de DfT, à travers l'ajout de ces wrappers, permet ainsi d'augmenter les taux de couverture de fautes. Dans ce type d'approche, le processeur envoie des paquets de test spécifiques, compréhensibles par le wrapper cible. La figure 7.2 en présente les concepts.

7.2.1 Des wrappers rendant les coeurs transparents

Papachristou [Papachristou99] s'est attaqué au problème en proposant une approche globale basée sur la réutilisation des interconnexions entre coeurs pour véhiculer les vecteurs de test. Afin de rendre les coeurs transparents entre les interconnexions, un mécanisme de *bypass* est ajouté à chaque coeur.

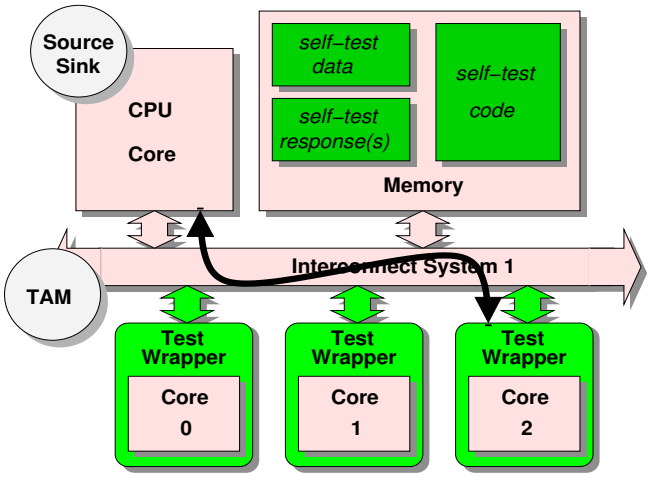


FIG. 7.2: Approches avec utilisation de wrapper de test

Cette circuiterie ad-hoc est décrite dans l'article [Nourani98]. Ainsi, cette technique permet de s'affranchir de la connexion requise avec l'interconnect système, contrairement aux approches purement fonctionnelle vues précédemment ([Jayaraman02] et [Tehranipour03]). De plus, cette technique permet de tester les interconnexions entre coeurs.

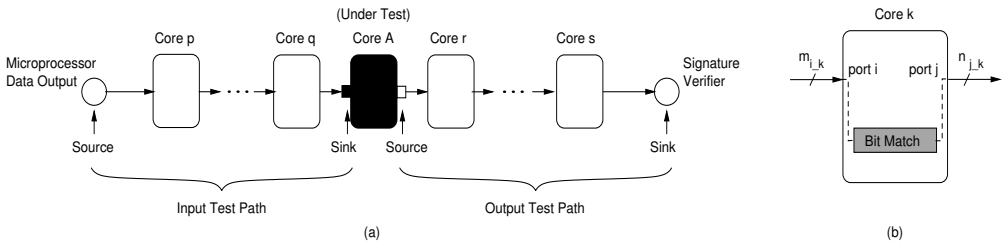


FIG. 7.3: L'approche "bypass" proposé par [Papachristou99]

Un registre de *bypass* permet de court-circuiter un coeur en connectant ses entrées sur ses sorties (figure 7.3 (b)). Etant donné la différence entre le nombre d'entrées et de sorties, le registre de *bypass* doit permettre une serialisation ou parallélisation des données ("bit-match circuit"). La circuiterie de *bypass* est complétée par de la logique permettant d'isoler le coeur pendant le transfert des vecteurs. Le microprocesseur embarqué configure le système avec l'aide d'un contrôleur de test simplifié, pour mettre certains coeurs en mode *bypass*. Ainsi, il existe un chemin permettant de véhiculer les vecteurs de test entre le processeur et le coeur d'un côté, et entre le coeur et un MISR compactant les réponses de l'autre (figure 7.3 (a)). La méthode est validée sur un exemple très petit (tutorial synopsys composé de 4 éléments de moins de 5000 fautes au total).

Les problèmes liés à cette approche sont :
- La solution est difficile à mettre en oeuvre et son implémentation doit être redéfinie pour chaque nouveau SoC. Cette approche ne se situe pas dans une optique de réutilisation du test.
- Elle nécessite l'introduction de matériel dédié non standard autour de chaque coeur.

– Elle nécessite l'existence ou la création d'un chemin de sortie (output test path) entre tous les coeurs et le MISR, ce qui peut engendrer des problèmes de routage.

7.2.2 L'architecture RASBuS : contrôle et observation des E/S

Hwang et Abraham proposent l'architecture RASBuS [Hwang01]. Le processeur accède aux registres internes du coeur en utilisant les primitives classiques de lectures/écritures sur le bus système. Ces valeurs déterminent le mode fonctionnel du coeur. Comme le processeur ne peut placer de valeurs précises sur les entrées ou bien capturer les valeurs de sorties, RASBuS ajoute un wrapper de test appelé "Test Access Interface" (TAI) contenant le registre de bordure (voir figure 7.4). De cette manière, le registre de bordure, mappé dans le système d'adressage, est directement accessible par le processeur, de la même façon que les autres registres internes du coeur.

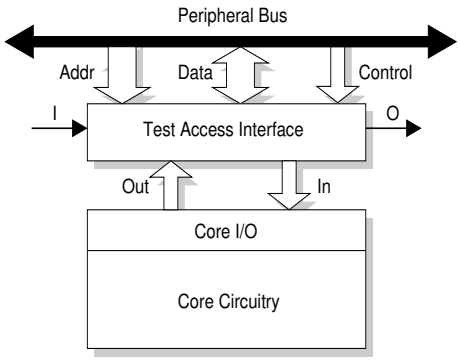


FIG. 7.4: Wrapper de l'approche RASBuS

Les problèmes liés à cette approche sont :
– L'interconnect est utilisé pour véhiculer les vecteurs de test, donc une connexion entre les coeurs testés et l'interconnect est obligatoire.
– L'ajout d'un wrapper de test spécifique, et donc non standard.
– La supposition que tous les registres internes sont adressables par le processeur, et, accessibles en lecture comme en écriture. Cette vision de l'accessibilité des registres internes rend cette approche très restrictive.

7.2.3 Des wrappers permettant la gestion du test At-Speed

Dans les articles [Huang01] et [Iyer02], les auteurs proposent une autre architecture de wrapper de test. Dans cette approche les auteurs décident de réutiliser les structures DfT présentent dans le coeur comme par exemple les chaînes de scan, basculant ainsi d'une approche fonctionnelle à une approche complètement structurelle. Le wrapper de test, connecté à l'interconnect système, reçoit les paquets de test délivrés par le processeur MIPS embarqué. Ces paquets de test contiennent les vecteurs à injecter dans le coeur. Grâce à un "buffer" interne au wrapper, stockant les vecteurs de test, et grâce à de la logique supplémentaire permettant le contrôle du chargement des vecteurs ("scan shifting"), le wrapper

peut appliquer les vecteurs de test au coeur. Les vecteurs de test peuvent être générés par le processeur lui-même ou récupérés depuis l'ATE externe et rangés dans la mémoire embarquée. L'utilisation des chaînes de scan pour le test, supprime la capacité de test *at-speed* fourni par une approche fonctionnelle. Pour pallier ce défaut, les auteurs proposent l'utilisation du schéma d'application appelé *Multiple capture Test Scheme* (MTS) présenté dans [Tsai00], permettant de réintégrer la composante *at-speed* du test du coeur. Un vecteur est chargé dans la chaîne de scan suivi de plusieurs (k) cycles fonctionnels de capture. Ainsi, en fonction du nombre d'entrées/sorties w ($w = max(PIs, POs)$) et pour k cycles de capture, un registre interne au wrapper de kw bits est utilisé pour stocker les réponses de test. La figure 7.5 représente l'architecture interne du wrapper de test. Les résultats reportés dans [Huang01] affichent des taux de couverture sur les fautes de collage de 92,5% à 99,9% sur les benchmarks ISCAS 89. Les taux de couverture révélés dans [Iyer02], sur les fautes de transition (transition faults) et les fautes de chemin (path delay faults) sont satisfaisants.

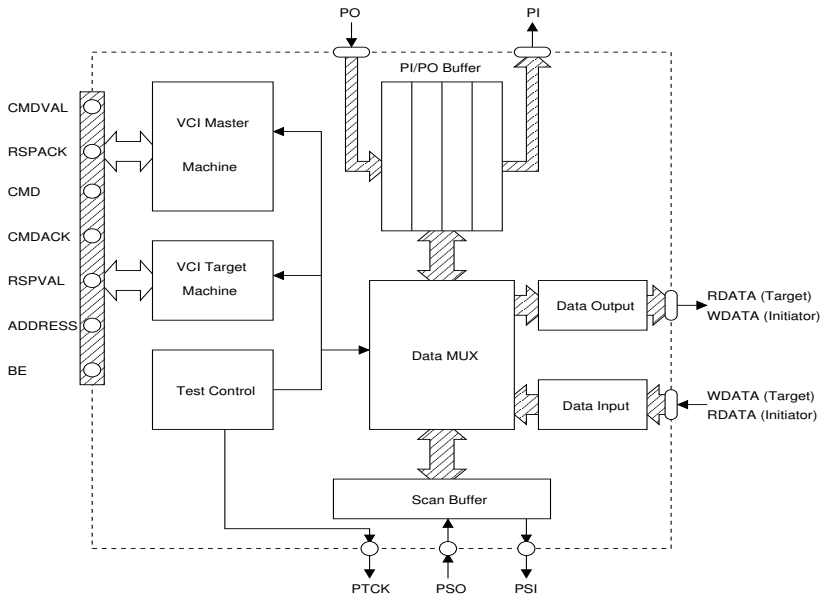


FIG. 7.5: Architecture interne du wrapper de test présenté dans [Huang01, Iyer02]

Les problèmes liés à cette approche sont :
- La connexion obligatoire du wrapper à l'interconnect système.
- L'ajout d'un wrapper de test spécifique, taillé sur mesure, et donc non standard.
- Les auteurs ne désirent pas communiquer sur la taille des wrappers. Cependant, on peut supposer que la surface engendrée par ces wrappers peut être importante, due au registre stockant plusieurs valeurs consécutives des entrées/sorties et au "buffer" interne stockant les vecteurs de test.
- Le format des paquets de test est intimement lié au protocole utilisé. Un des rôles du wrapper de test est de décoder les paquets transmis sur l'interconnect. Il doit donc être reconçu en fonction de l'interconnect utilisé.
- La gestion du test *at-speed* par MTS. Le problème est que ce type de génération n'est pas standard (non géré par les ATPGs).

7.3 Approches avec micro-testeurs embarqués

Allant plus loin dans l'insertion de DfT, certains chercheurs se sont penchés sur l'ajout de micro-testeurs intégrés dans la puce. Ces micro-testeurs sous le contrôle du microprocesseur embarqué délivrent les vecteurs de test aux coeurs wrappés auxquels ils sont connectés. L'ajout de micro-testeurs permet de réduire certains défauts des approches précédentes qui n'utilisent que des wrappers de test. En particulier, des wrappers standards (comme les wrappers de test IEEE 1500) peuvent être pilotés par ces micro-testeurs. La figure 7.6 présente l'utilisation de micro-testeurs dans un SoC.

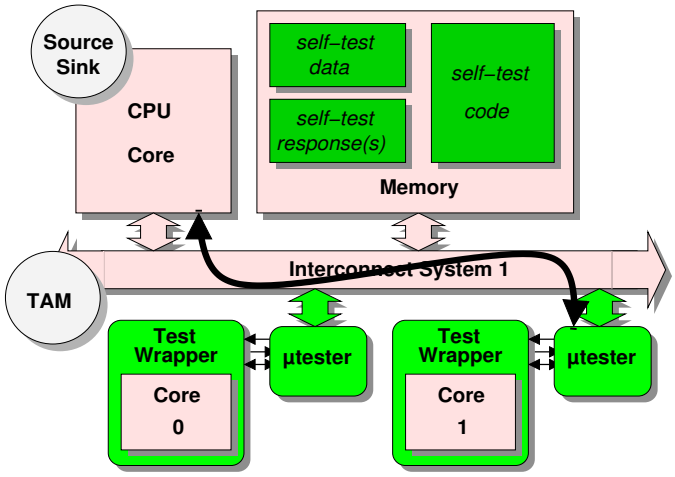


FIG. 7.6: Approches avec utilisation de wrapper de test

7.3.1 Approche clients/serveur

L'approche proposée dans [Wang04] repose sur l'utilisation de deux types de micro-testeur : le serveur et le client. Le serveur délivre les paramètres de test aux différents clients, sous la forme de paquets transitant sur l'interconnect système. Ainsi, chaque client est connecté à l'interconnect système et est adressable par le serveur. De l'autre côté le micro-testeur client accède aux ports de test d'un ou de plusieurs coeurs (entrées/sorties des chaînes de scan et ports d'accès au boundary scan (TAP) par exemple). La figure 7.7 présente ces différents aspects. Les résultats en terme de surface additionnelle et de temps de test sont reportés sur trois SoC (SOC1, SOC2 et SOC3), mappés sur FPGA. Ces trois SoCs sont conçus avec des coeurs correspondants aux benchmarks ITC 99. Les auteurs précisent que la surface additionnelle liée aux micro-testeurs ajoutés (un par composant testé) représente de 2% à 7% de la surface totale des différents SoCs.

Les problèmes liés à cette approche sont :

– La surface additionnelle reportée est une évaluation sur FPGA. De plus, les auteurs nous précisent la taille supplémentaire sur les SoCs entiers. Néanmoins, les résultats présentés nous permettent d'évaluer la surface ajoutée pour chaque coeur. Pour le SOC1, la surface ajoutée pour chaque coeur, est en moyenne de 16.10%, pour le SOC2 elle est de 3.66% et enfin pour le SOC3, de

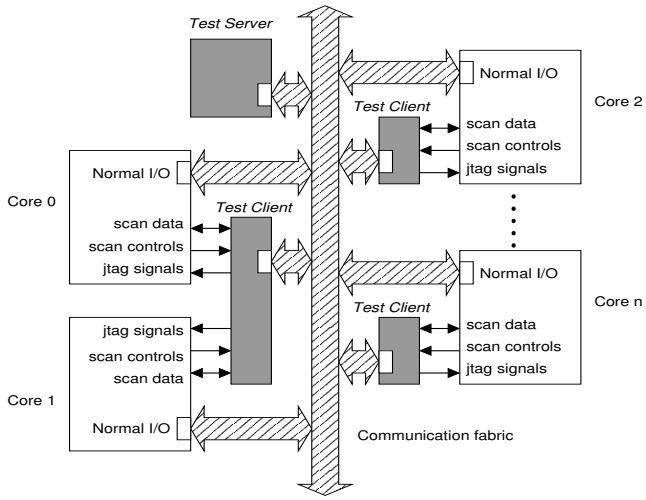


FIG. 7.7: L'approche serveur/client de test.

12.25%. Ainsi, sur l'ensemble des trois SoCs présentés, les micro-testeurs ajoutent en moyenne par coeur testé, 10.7% de surface additionnelle.

– L'architecture proposée pour le micro-testeur est présentée comme étant modulaire. Chaque client de test peut être connecté à plusieurs coeurs différents. Cependant, les résultats sont donnés sur des SoCs ayant un micro-testeur par coeur testé. Or une connexion sur un interconnect système implique de la circuiterie supplémentaire. En effet, si l'interconnect est un bus, le contrôleur est alors complexifié. Si l'interconnect est un bus de type AMBA AHB ou un cross-bar, l'ajout d'une connexion est très coûteuse puisque les connexions se font à base de multiplexeurs. Un tel ajout de connexions sur certains types d'interconnects peut induire un supplément de circuiterie non négligeable.
– Tous les clients sont en attente des paquets de test et des commandes émis par le serveur. Le serveur est donc le goulot d'étranglement de la méthode.
– L'utilisation des chaînes de scan et/ou du Boundary-Scan supprime la capacité de test *at-speed* fournie par les approches fonctionnelles. Or aucun mécanisme de test *at-speed* n'est explicitement géré.
– L'approche repose sur la génération pseudo-aléatoire de vecteurs de test. Aucune gestion des vecteurs de test déterministes fournis avec les coeurs n'est présentée, ce qui peut poser problème lorsque le coeur est fourni sous forme de *hard IP*.

7.3.2 Le processeur de test MET

Erika Cota et al. [Cota01] se placent dans le contexte du test d'un SoC contenant des *hard IP*. Les auteurs proposent l'utilisation d'un processeur dédié au test embarqué des coeurs entourés d'un wrapper standard (Boundary-Scan ou IEEE 1500) et fournis avec les vecteurs de test déterministes au format CTL ou STIL. Le processeur de test MET (Microprocessor for Embedded Test) est ajouté dans la puce. Les mémoires embarquées du SoC sont réutilisées pour stocker le programme de test et les

vecteurs de test. MET peut commander le test d'un coeur ayant des interfaces standards comme l'interface JTAG et l'interface IEEE 1500, voire un contrôleur BIST. Le programme de test exécutable par MET provient de la compilation du fichier CTL ou STIL fourni avec le coeur. Le jeu d'instruction MET contient une dizaine d'instructions. Une évaluation sur FPGA de la taille de MET est donnée. Les comparaisons en terme de surface montrent que le processeur ajoute 15% de surface pour un processeur RISC 32 bits.

Les problèmes liés à cette approche sont :
– Aucun renseignement n'est donné sur la stratégie du test au niveau système.
– Aucune valeur numérique n'est donnée concernant les temps de test de MET.
– MET utilise les vecteurs déterministes fournis avec l'IP. La stratégie suppose qu'ils sont stockés dans des mémoires internes au moment du test. Or le chargement dans le SoC d'un telle quantité de données par un équipement de test pèse sévèrement sur le temps de test.
– La fréquence de test de MET est au maximum égale à la moitié de la fréquence fonctionnelle de MET.

7.3.3 Une plate-forme de test

Les auteurs de l'article [Lee05], présentent un micro-testeur embarqué capable de tester des coeurs wrappés à la norme IEEE 1149.1 ou IEEE 1500. Il peut contrôler des BIST pour mémoires et des BIST pour composants analogiques. Le processeur embarqué contrôle l'exécution du processus de test. Le micro-testeur (appelé "TAM Controller" dans l'article) se charge de l'exécution du test de chaque composant. Tous les coeurs sont connectés à un bus de test contrôlé par le "TAM controller". Le figure 7.8 présente la plate-forme utilisée. Les vecteurs utilisés sont ceux fournis avec l'IP. De par la taille que représente ce volume de données, les auteurs ont décidé de stocker les vecteurs de test et les réponses attendues dans une mémoire externe. L'exécution du processus de test s'effectue selon 3 points qui sont itérés tant qu'il reste des vecteurs de test à appliquer :

1. Le processeur embarqué charge un bloc de données de test depuis la mémoire externe dans la mémoire interne.
2. Le micro-testeur effectue alors le test du coeur considéré.
3. Le processeur vérifie les réponses. Si les réponses sont correctes, le processus est ré-exécuté à partir du point 1.

Le micro-testeur contient un MISR chargé de compacter les réponses et d'accélérer le processus de test.

La plate-forme utilisée pour mesurer les performances du micro-testeur contient une dizaine de coeur. La taille des données de test est de 0.754 Mbits, ces données formatées pour le micro-testeur occupe une place de 1.106 Mbits, soit une augmentation d'environ 35%. La surface additionnelle engendrée par cette plate-forme de test est de 8.05%. Finalement, le nombre de cycles nécessaire aux tests de ces 10 coeurs est d'environ 5 millions de cycles.

Les problèmes liés à cette approche sont :
– Le contrôleur supporte la norme IEEE 1149.1 en natif. Des traducteurs de protocole doivent être ajoutés devant chaque wrapper IEEE 1500 afin de faire la traduction entre ces deux protocoles.
– Le bus de test est partagé entre les différents coeurs testés. Un seul coeur peut être testé à la fois.

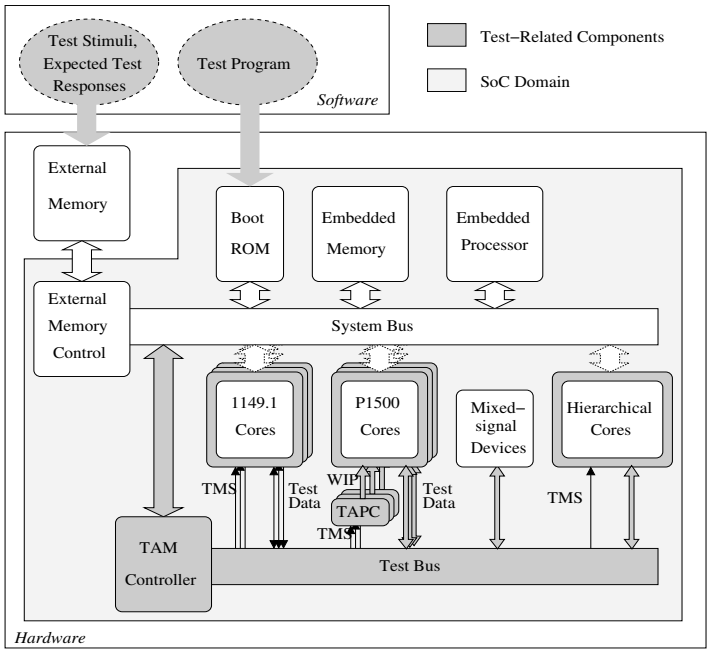


FIG. 7.8: La plate-forme de test présentée dans [Lee05].

– Le processeur embarqué est utilisé pour chercher les données de test depuis l'extérieur, les formater et les stocker en mémoire interne. Ainsi, la majeure partie des transferts est initiée par le processeur embarqué et n'est pas utile au test. Autrement dit, le processeur est donc le goulot d'étranglement de la méthode.

7.4 Conclusion

Nous avons pu distinguer trois types de méthodes génériques pour l'auto-test logiciel des SoCs.

7.4.1 Approches purement fonctionnelles

La première, propose une approche complètement fonctionnelle. Bien qu'ayant l'avantage de n'ajouter aucune surface supplémentaire dédiée au test, la contrepartie est le faible taux de couverture et une réutilisation du test quasiment inexistante. Globalement, voici les points faibles des approches purement fonctionnelles :

- Nécessité de connaître la fonctionnalité de tous les coeurs testés.
- Programme de test dépendant du système.
- Taux de couvertures faibles.
- Connexion entre le coeur et l'interconnect système obligatoire.

7.4.2 Approches avec wrapper de test

Introduisant de la DfT, les approches nécessitant l'utilisation de wrappers de test autour des coeurs et réutilisant les chaînes de scan permettent d'améliorer les taux de couverture. Néanmoins, l'introduction d'un wrapper amène les points faibles suivants :

- Nécessité d'utiliser des wrappers de test spécifiques et donc non standard.
- Une connexion entre le wrapper et l'interconnect système est obligatoire.
- Les wrappers contiennent des buffers stockant la totalité d'un vecteur de test, la surface engendrée par ces wrappers est donc très importante.
- La seule gestion du test *at-speed* proposée, repose sur l'utilisation de la technique MTS, qui n'est pas standard.

7.4.3 Approches avec micro-testeurs embarqués

L'ajout d'un micro-testeur embarqué permet l'augmentation de la réutilisation du test à travers l'utilisation de wrappers standards. Les approches proposées jusqu'ici souffrent de certains points :

- Introduction de surface supplémentaire due au micro-testeur.
- Les coeurs ne sont pas testés en parallèle.
- Pas de mécanisme de gestion de test *at-speed* explicite (AC scan).

Le tableau 7.1 résume les différentes caractéristiques des approches proposées. Des trois approches, l'approche par micro-testeur semble être celle permettant d'obtenir des coûts de test les plus bas. En effet, cette approche permet une réutilisation des composants élevée, s'intègre dans un flot classique par l'utilisation de wrappers standards et permet d'aboutir à une qualité de test élevée de par l'utilisation de vecteurs calculés par ATPG.

	Fonctionnel	**Wrapper**	**Wrapper + μtesteur**
Connaissance du coeur	élevée	moyenne	faible
Taux de couverture	faible	moyen	élevé
Test At-Speed	fonctionnel	fonctionnel [Hwang01] structurel [Huang01, Iyer02]	non
Génération vecteurs	on-chip	on-chip	off-chip/on-chip stockage interne [Cota01] stockage externe [Lee05]
Type de wrapper	-	non standard	standard
Connexion wrapper/coeur à l'interconnect	requise	requise	non requise
Surface supplémentaire	nulle	élevée	moyenne
"Test reuse"	faible	moyen	élevé

TAB. 7.1: Caractéristiques des différentes approches génériques pour le test des SoCs.

Chapitre 8

Stratégie proposée

La norme IEEE 1500 permet une intégration aisée, dans un SoC, des coeurs compatibles. Ainsi, elle permet de réduire les coûts du test. L'auto-test logiciel (SBST), de par son aspect embarqué, permet quant à lui l'utilisation d'équipements de test externe bas coût. Son aspect programmable, contrairement à l'auto-test matériel (BIST), rend la conception du test flexible.

Ainsi notre objectif a été de concevoir une méthode basée sur le SBST, permettant de tester des coeurs munis d'un wrapper standard. L'étude de l'état de l'art nous a permis de constater que l'utilisation de micro-testeurs embarqués était la seule stratégie SBST permettant le test des coeurs compatibles avec la norme IEEE 1500. En effet, l'interface de test du wrapper ne peut être accédée directement via un interconnect système standard. C'est pourquoi nous proposons une stratégie de test basée sur l'utilisation d'un micro-testeur qui, tout en gardant les avantages du SBST, réduit voire supprime les inconvénients des approches citées précédemment. Le micro-testeur développé a les caractéristiques suivantes :

- Utilisation des vecteurs de test déterministes fournis avec l'IP.
- Test concurrent des coeurs.
- Temps de test minimal.
- Conception modulaire afin de faciliter la gestion d'autres protocoles de test.
- Surface additionnelle réduite.
- Test *at-speed* standard (géré par ATPG) des coeurs.

Dans ce chapitre nous étudierons la stratégie de test proposée. Nous verrons quels composants doivent être présents dans le SoC, pour que notre stratégie puisse être implémentée. Nous présenterons une architecture de micro-testeur modulaire permettant de pallier certains défauts présentés dans l'état de l'art. L'utilisation du processeur embarqué nous permettra de dessiner une stratégie de test globale, au niveau système. Les différents formats de programmes de test, ceux exécutés par le micro-testeur et celui exécuté par le processeur embarqué, seront décrits dans la dernière partie.

8.1 Les systèmes sur puces visés

La méthode de test proposée peut s'appliquer aux SoCs ayant les caractéristiques représentées sur la figure 8.1. Notre stratégie fait l'hypothèse que le SoC à tester possède certains composant matériels. Bien que ces composants matériels soient propres à une majorité de SoCs actuels, il est intéressant et nécessaire de les présenter. Notre stratégie repose aussi sur l'utilisation de quelques programmes de test, ce sont les composants logiciels. Nous allons présenter dans la première partie de cette section les composants matériels requis, puis la seconde partie abordera les composants logiciels.

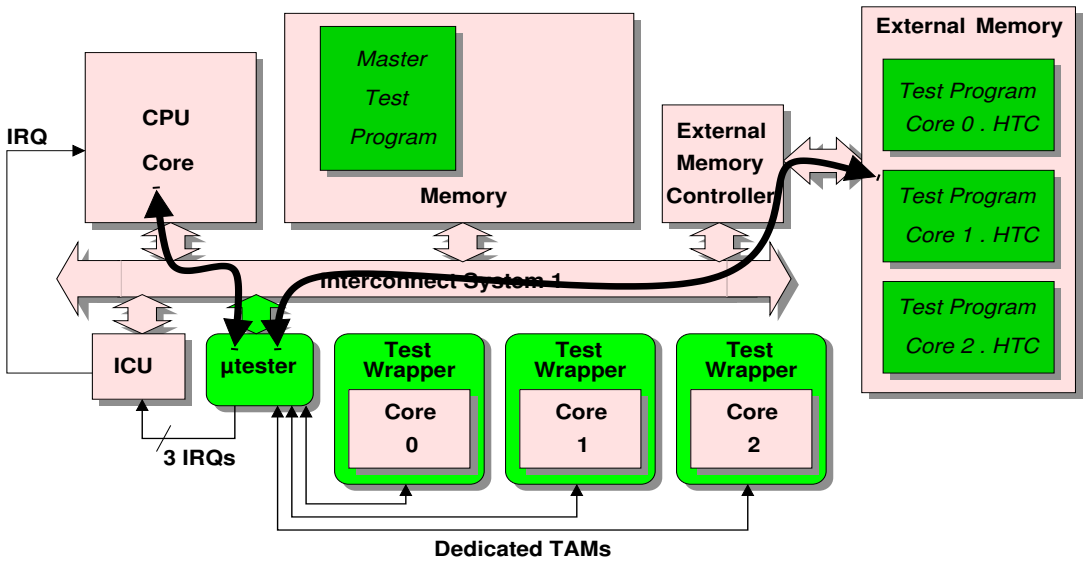


FIG. 8.1: Composants matériels et logiciels nécessaires dans un SoC à notre stratégie.

8.1.1 Les composants matériels

Tout d'abord voyons quels sont les composants nécessaires et que l'on trouve classiquement dans les stratégies SBST. Le SoC doit contenir un processeur embarqué permettant d'exécuter le programme de test. Ce dernier est stocké dans une mémoire interne. Le processeur doit pouvoir accéder à la mémoire grâce à un interconnect système supportant le schéma initiateur/cible.

Nous nous sommes placés dans un contexte où les coeurs délivrés sont compatibles avec la norme IEEE 1500. Ainsi, chaque coeur est muni d'un wrapper de test compatible avec la norme.

Les stimuli à appliquer sont des vecteurs de test déterministes fournis avec les coeurs à tester. Puisque cet énorme quantité de données ne peut être stockée dans la puce, nous décidons de les ranger en dehors de la puce, dans une RAM externe. Ainsi, nous supposons que le SoC est équipé d'un contrôleur de RAM externe. Lorsque le SoC est utilisé en mode fonctionnel, le contrôleur de RAM externe est utilisé pour brancher une mémoire externe ou un périphérique. Lorsque le SoC est utilisé en mode test, les plots d'entrées/sorties de l'interface sont connectés à une mémoire externe contenant les programmes de test. Comme nous le détaillerons plus tard, chaque coeur à tester possède son propre

programme de test stocké dans la mémoire externe. Il existe donc dans la RAM externe, autant de programmes de test que de coeurs à tester. Les programmes de test sont exécutables par le micro-testeur, dans un format spécifique que nous avons appelé HTC (Hexa Test Code).

Une unité de contrôle des interruptions (Interrupt Control Unit ICU) est utilisée afin de centraliser les requêtes d'interruptions (Interrupt Request IRQ) venant du micro-testeur embarqué. Dans le cas où seulement quelques coeurs sont testés, et qu'il y a autant d'IRQ disponibles sur le processeur, les IRQs du micro-testeur peuvent être directement branchées sur le processeur.

A côté de ces composants que nous trouvons usuellement dans les SoCs, un nouveau composant matériel est ajouté, entièrement dédié au test du SoC : le micro-testeur.

Architecture du micro-testeur

Nous désirons minimiser les connexions avec l'interconnect système car nous avons vu que sur certains interconnects l'ajout de connexions introduit beaucoup de matériel supplémentaire. Ainsi nous optons, dans un premier temps, pour l'utilisation d'un seul micro-testeur dans le SoC. Nous désirons également minimiser le temps de test, le micro-testeur doit donc supporter le test concurrent de plusieurs coeurs. Le micro-testeur possède donc deux interfaces principales.

D'un côté, il s'interface avec le système via l'interconnect système. Il est alors localisé dans l'espace d'adresses mémoire du système et peut donc être adressé comme n'importe quelle ressource par le processeur embarqué. En tant que cible, il peut recevoir les commandes émises par le processeur, en tant qu'initiateur il peut accéder directement à la RAM externe et ainsi, lire les programmes de test qui y sont rangés.

De l'autre côté, il est branché aux coeurs à tester, par le biais de TAMs dédiés à chaque coeur. Ainsi il peut contrôler, envoyer et recevoir les vecteurs de test sur tous les coeurs en parallèle.

Comme le montre la figure 8.2 le micro-testeur est composé de deux types de composants : les TPIUs et le Prefetch-Buffer.

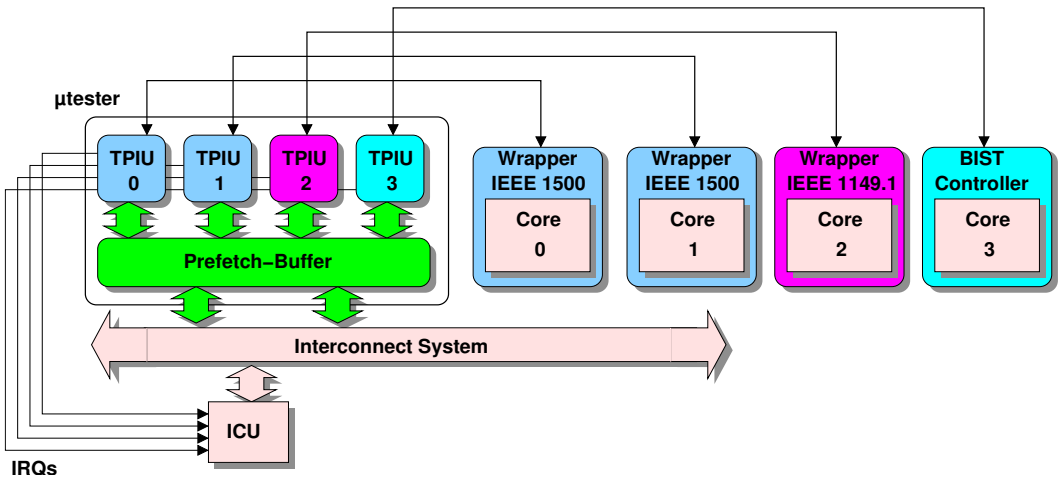


FIG. 8.2: Architecture interne du micro-testeur : une conception modulaire.

Les TPIUs. Les TPIUs sont les composants qui exécutent les programmes de test. Il y a un TPIU dédié à chaque coeur testé. L'ajout d'un nouveau coeur à tester se fait simplement en "branchant" un nouveau TPIU sur le Prefetch-Buffer et en reliant ce TPIU au coeur, par le biais d'un TAM dédié. Cette conception complètement modulaire du micro-testeur permet de le rendre "scalable". La surface sur silicium du micro-testeur est proportionnelle au nombre de coeurs testés. Chaque TPIU est compatible avec la stratégie de test du coeur auquel il est connecté. Il existe des TPIUs permettant de conduire le test de coeurs à la norme IEEE 1500, des coeurs compatible Boundary Scan, et des coeurs "bistés". Ainsi, le micro-testeur n'est pas dédié à un protocole de test en particulier. Néanmoins, dans la suite de ce manuscrit nous ne nous intéresserons qu'à la norme IEEE 1500, puisque c'est cette norme qui régit de plus en plus l'intégration des coeurs dans les SoCs. Les TPIUs agissent de façon complètement indépendante les uns des autres, permettant ainsi le test concurrent des coeurs. Chaque TPIUs peut alerter le processeur via une IRQ, pour l'avertir que le test du coeur est terminé, soit parce qu'une erreur a été détectée, soit parce que le test complet s'est déroulé correctement.

Tous les TPIUs sont connectés au même composant : le Prefetch-Buffer (PB). Le PB établit la communication entre le système et les TPIUs. Les TPIUs sont donc complètement indépendants du protocole utilisé par l'interconnect système.

Le Prefetch-Buffer. Le rôle défini pour le PB est de fournir une interface entre l'interconnect système et les TPIUs. Le PB est connecté à l'interconnect système par deux ports : un port initiateur et un port cible.

A travers le port initiateur le PB agit comme un DMA en récupérant des fragments de programmes de test depuis la RAM externe du SoC. Le PB est une sorte de mémoire cache, contenant des bouts de programmes de test, permettant de minimiser le temps d'oisiveté des TPIUs.

A travers le port cible le micro-testeur peut recevoir des commandes du processeur. Le PB gère l'espace d'adressage complet du micro-testeur. Lorsque le processeur désire communiquer avec un TPIU en particulier, le PB établi un canal de communication entre le processeur et le TPIU visé. Ainsi, le processeur peut converser avec n'importe quel TPIU de façon indépendante.

Pour résumer, l'utilisation du Prefetch-Buffer permet de rendre les TPIUs complètement indépendants du protocole de communication utilisé par l'interconnect système. De plus, l'effet "cache" permet de minimiser l'oisiveté des TPIUs et de les faire fonctionner en parallèle.

8.1.2 Les composants logiciels

Dans notre stratégie le processeur embarqué s'occupe du processus de test au niveau système, le micro-testeur quant à lui s'occupe du test au niveau coeur. Il existe donc deux types de programme de test.

Le premier, appelé "programme de test maître" (MTP) est exécuté par le processeur embarqué. Il est chargé au démarrage dans la mémoire interne du SoC, mais peut aussi résider en mémoire externe.

Le deuxième type de programme, correspond au programme de test exécuté par le micro-testeur. Les programmes de test, un par coeur à tester, sont décrits dans un format spécifique appelé HTC, compréhensible par le micro-testeur. Ces programmes HTC résident en mémoire externe.

Le programme de test maître

Le programme de test maître exécuté par le processeur embarqué supervise l'exécution du processus de test global. Ce programme maître peut être écrit en assembleur ou dans des langages plus haut niveau comme le langage C. Ensuite le programme est (cross-)compilé et le binaire résultant est chargé dans la mémoire interne du SoC ou peut être stocké dans la mémoire externe. La facilité d'écriture dans des langages haut niveau permet de concevoir un programme de test maître aussi complexe que désiré : plan de test intelligent, redémarrage des programmes de test sur des parties spécifiques, ou même générer des programmes HTC à la volée. Le processeur peut interroger le micro-testeur (chaque TPIU) afin de connaître l'état du test de chaque coeur (en cours, échec, succès etc.). Afin d'éviter de surcharger l'interconnect système il peut attendre les interruptions émises par le micro-testeur. Un exemple de programme de test est donné en annexe D.4.

Les programmes de test HTC

Les programmes de test HTC sont exécutés par le micro-testeur (par les TPIUs pour être plus exact). Chaque coeur testé dispose de son propre programme HTC stocké en mémoire externe. Ce programme contient principalement les stimuli à appliquer, les réponses attendues à comparer ainsi que les instructions de configuration du TPIU. Un programme HTC est une séquence d'instructions de test de 32 bits. L'exécution d'un tel programme est complètement séquentielle, sans boucle ni saut. Une instruction de test est divisée en deux champs : 6 bits d'opcode et 26 bits pour différents paramètres. Une instruction de test est suivie par zéro ou plusieurs mot de 32 bits contenant les données de test, comme montré sur la figure 8.3. Les données de test sont les stimuli ou les réponses attendues. Lorsque les vecteurs (stimuli ou réponses) font plus de 32 bits, chaque vecteur est découpé en plusieurs tronçons de 32 bits. Si la taille du vecteur n'est pas un multiple de 32, au dernier tronçon seront ajoutés des bits de bourrage.

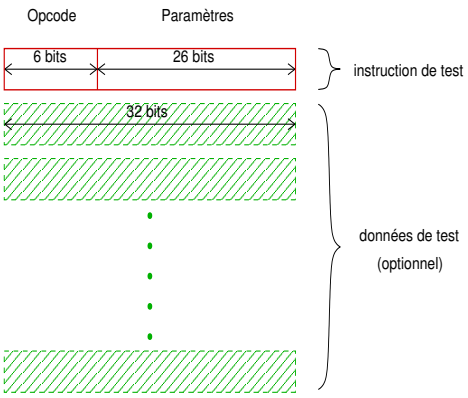


FIG. 8.3: Un programme de test HTC est une séquence d'instructions de 32 bits. Une instruction peut être suivie par des mots de données de 32 bits optionnels.

Les programmes de test HTC peuvent être générés depuis le fichier STIL/CTL fourni avec le coeur. Afin d'écrire facilement ces programmes, une bibliothèque appelée STELA a été développée en lan-

gage C++. L'annexe D.3 présente un exemple d'utilisation de la bibliothèque STELA ainsi que l'outil GenSTELA.

8.2 Exécution du processus de test

Après cette vue panoramique des composants matériels et logiciels, nous allons voir dans cette section comment tous ces composants sont utilisés, afin d'effectuer le processus global de test dans la puce.

Le processus de test consiste en une collaboration entre le processeur et le micro-testeur. Le micro-testeur est chargé d'exécuter les programmes de test de chaque coeur. Le processeur supervise le processus global de test dans la puce.

Nous allons tout d'abord présenter les différentes étapes du processus global de test. Nous verrons alors que deux modes de test s'offrent à nous : le mode test de production, et le mode diagnostic. La deuxième partie de cette section se concentrera sur le mode diagnostic.

8.2.1 Description du processus global de test

L'exécution du processus de test peut être découpé en 5 étapes principales :

Étape 0 : Test des composants utilisés

Notre stratégie repose sur l'utilisation de composants embarqués dans la puce, comme le processeur, la mémoire interne ou l'interconnect système. Le micro-testeur ne testant pas ces composants, voyons comment ils peuvent être testés avant d'être utilisés.

Le processeur embarqué peut être testé de façon logicielle. Les techniques SBST des processeurs ont été traitées dans le début de ce manuscrit, néanmoins nous pouvons proposer l'utilisation des techniques proposées dans [Gizopoulos04] qui permettent d'obtenir des taux de couverture élevés (supérieur à 90%) avec des programmes de test très légers (de l'ordre du kilo-octets).

La mémoire interne peut être testée par un mécanisme matériel (BIST) comme logiciel (SBST). [Rajsuman99] et [Tehranipour01] proposent d'utiliser le processeur embarqué pour tester les mémoires.

Le processeur embarqué peut être aussi utilisé pour tester l'interconnect système. Les interconnects systèmes contiennent de longs fils parallèles qui peuvent interférer entre eux à cause des capacités de couplages inter-fils, des inductances mutuelles etc. Les articles [Chen01a, Chen02] et [Lai01] abordent le sujet et décrivent des techniques logicielles permettant de détecter les effets dus à la diaphonie (cross-talk).

Après le test de ces composants, le processus de test des coeurs wrappés peut commencer.

Étape 1 : Démarrage du processus de test

Le processus de test commence de la même manière que les autres stratégies SBST. Le programme de test du processeur embarqué est chargé dans la mémoire du SoC par un équipement de test externe. Lorsque le programme de test est chargé en mémoire, un "reset" du processeur est alors effectué afin qu'il débute l'exécution.

Le rôle dévoué au processeur est celui du chef d'orchestre. Il est en charge de déclencher le test des coeurs désirés. Pour ce faire, le processeur envoie au micro-testeur l'adresse à laquelle se trouve le programme de test HTC à exécuter (situé en mémoire externe), ainsi qu'une requête de démarrage du test. Le micro-testeur va donc aller chercher le programme de test situé en mémoire externe et l'exécuter. Le processeur peut lancer ainsi le test sur tous les coeurs, afin que les tests s'exécutent en parallèle.

Une fois les tests lancés, le processeur se place en attente d'interruptions.

Étape 2 : Exécution des programmes de test par le micro-testeur

Le micro-testeur (le Prefetch-Buffer plus exactement) détient donc les adresses des différents programmes de test, envoyées précédemment par le processeur (étape 1). Le Prefetch-Buffer va alors récupérer des fragments de programmes de test et les stocker dans des tampons mémoires. Chaque TPIU dispose de son propre tampon mémoire dans le Prefetch-Buffer. Etant donnée la structure des programmes HTC (complètement séquentiel, sans boucles ni sauts), les tampons mémoires utilisés par le PB sont des FIFOs.

L'exécution du programme de test par le TPIU consiste à décoder la suite d'instructions stockées dans la FIFO, correspondant à l'application du vecteur, la capture de la réponse, la récupération et le stockage des réponses dans le micro-testeur.

Étape 3 : Comparaison des réponses

Lorsque le TPIU a ramené les réponses du coeur il les compare à celles attendues. A ce niveau deux stratégies peuvent être employées : le test de production et le diagnostic.

En test de production, le test doit être effectué le plus rapidement possible. La seule information nécessaire consiste à savoir si le test de la puce a échoué ou non. Pour cela, le TPIU est configuré en mode test de production : un MISR de 32 bits est utilisé pour compacter les réponses venant du coeur testé. Cette technique permet une réduction drastique de la taille des programmes de test. En effet, toutes les réponses attendues sont remplacées par une unique signature sur 32 bits.

Par contre, en mode diagnostic l'objectif est différent, il est nécessaire de connaître les bits fautifs de la réponse donnée par le coeur. Pour ce faire, le mode compaction est désactivé, et les réponses sont comparées avec celles attendues bit à bit.

Nous pouvons signaler le fait que le test avec le micro-testeur, même en mode test de production, nous permet de localiser de façon approximative un défaut dans la puce, puisque nous savons quels sont les coeurs fautifs.

Étape 4 : Fin du test

Lorsque que le TPIU détecte une erreur (comparaison différente), ou que le test est terminé sur le coeur, le TPIU déclenche une IRQ. Le processeur interroge l'ICU afin de récupérer le numéro du TPIU qui a lancé l'IRQ. Le processeur consulte alors le TPIU concerné, afin de savoir si le test s'est déroulé correctement pour le coeur ou si une erreur a été détectée.

Le processeur récupère ainsi pour chaque coeur, le statut du test. Une fois ces informations récupérées, le processeur peut ranger les résultats, soit dans la RAM externe, soit dans un composant avec lequel l'équipement de test externe peut dialoguer [Bernardi04].

8.2.2 Le processus de test en mode diagnostic

Précisons maintenant ce que nous entendons par "diagnostic" et présentons la méthodologie employée.

Le fichier FDR

Dans un processus de test traditionnel conduit par un ATE, lorsqu'un composant échoue au test, le testeur peut ranger dans un fichier les informations concernant la panne. Nous appellerons ce fichier de pannes le fichier FDR (Failure Data Report).

De manière générale, le fichier FDR contient pour chaque faute, le numéro du vecteur qui a échoué, suivi du nom du port de sortie de la chaîne de scan (par exemple dans un coeur muni d'un wrapper IEEE 1500 : WSO), puis nous devons ajouter la position de la cellule dans la chaîne qui a capturé la donnée erronée. Ainsi les informations à fournir sont donc : le numéro de vecteur fautif, le nom du port de sortie de la chaîne de scan et enfin le numéro de la cellule fautive.

Grâce aux trois fichiers que sont, le fichier VHDL contenant la netlist, le fichier STIL/CTL contenant les vecteurs de test et le fichier de pannes FDR, un simulateur de faute avec des fonctionnalités de diagnostic peut déterminer la cause des vecteurs fautifs et générer un rapport de diagnostic, i.e. quel net est collé à 0 ou collé à 1 (figure 8.4).

Génération du fichier FDR à l'aide du micro-testeur

Afin de ne pas avoir recours à un ATE pour cette phase de diagnostic, nous avons doté le micro-testeur de certaines capacités afin de pouvoir récupérer les informations nécessaires au diagnostic.

Dans le mode diagnostic, le programme HTC ne contient pas de signature attendue mais toutes les réponses attendues bit par bit. Le compacteur du TPIU sélectionné est désactivé et agit comme un registre à décalage classique où les réponses données par le coeur testé sont comparées bit par bit avec celles attendues. Pour le moment, l'implémentation du mode diagnostic ne gère seulement que les coeurs équipés d'un wrapper IEEE 1500 à interface série. Dans ce cas précis, le nom du port scan-out est connu : Wrapper Serial Output (WSO). Le micro-testeur doit donc récupérer le ou les numéros des vecteurs fautifs et le ou les numéros des cellules fautives.

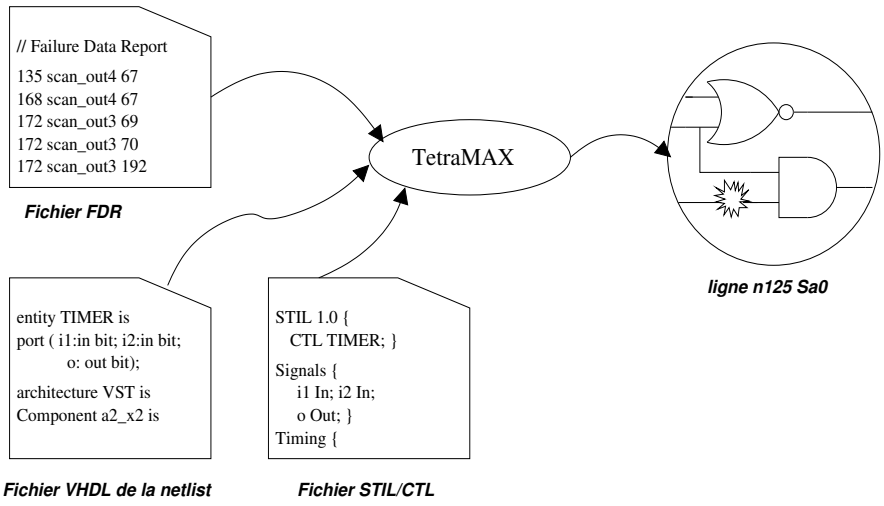


FIG. 8.4: Fichiers nécessaires pour effectuer un diagnostic.

Le mode diagnostic développé doit être effectué en deux phases. La première phase permet d'obtenir le numéro du vecteur fautif. La deuxième phase a pour but d'obtenir la position de la cellule interne de la chaîne de scan.

Phase 1 : identification des vecteurs fautifs. Le numéro du vecteur fautif est obtenu grâce à un registre interne spécifique dans le TPIU appelé MARKER. Dans la première phase, avant l'exécution de chaque vecteur de test, nous enregistrons, grâce à une instruction HTC spécifique, le numéro du vecteur en cours de traitement dans le registre MARKER. Ainsi, si le vecteur est fautif, une comparaison erronée lèvera l'IRQ du TPIU. Le processeur est alerté et peut récupérer la valeur rangée dans le registre MARKER du TPIU appelant. Le numéro du vecteur fautif est alors récupéré.

Phase 2 : identification des cellules fautives. Après cette première phase, nous possédons le numéro de tous les vecteurs fautifs. Dans la seconde phase une procédure quasiment identique est appliquée pour récupérer la ou les cellule(s) fautive(s). Dans cette phase, nous réutilisons le registre MARKER afin de marquer chaque bit comparé. Avant la première comparaison, le registre MARKER est initialisé à la valeur 0. Le TPIU récupère un bit et le compare à celui attendu. Puis la valeur du MARKER est incrémentée. Le TPIU récupère alors un deuxième bit et le compare à celui attendu, etc. Lorsqu'un bit est erroné, le TPIU active une interruption, le processeur embarqué relève ensuite, dans le registre MARKER, le numéro du bit fautif.

Toutes ces étapes du diagnostic sont implémentées en logiciel afin d'alléger la partie matérielle du micro-testeur. En annexe D.5, un exemple de déroulement du diagnostic est présenté.

La taille du MARKER est de 16 bits permettant en phase 1 de différencier 65536 vecteurs, et en phase 2, de pouvoir sonder une chaîne de scan de 65536 bits.

8.3 Conclusion

Ce chapitre nous a permis de présenter la stratégie de test proposée. Basée sur l'utilisation d'un micro-testeur embarqué dans le SoC, cette stratégie permet de tester les coeurs compatibles avec la norme IEEE 1500.

Le test de la puce est effectué par le couple processeur/micro-testeur. Le processeur est en charge du processus de test au niveau système, pendant que le micro-testeur se préoccupe du test au niveau coeur. Les vecteurs de test fournis avec le coeur sont encapsulés dans un programme de test exécutable par le micro-testeur : les programmes HTC. Ainsi chaque coeur dispose de son propre programme de test stocké en mémoire externe. L'exécution du programme de test consiste en l'envoi des stimuli, la capture des réponses et la comparaison avec celles attendues. Deux modes de test sont disponibles : le mode test de production, où les réponses sont compactées et où une signature est comparée à la fin du test et le mode diagnostic où les réponses récupérées sont comparées bit à bit aux réponses attendues. L'architecture complètement modulaire du micro-testeur permet de tester des coeurs de façon concurrente. Bien que la norme IEEE 1500 ait été prise comme exemple, le micro-testeur peut adresser tout autre protocole de test.

Chapitre 9

Aspects matériels et temporels

Le chapitre précédent a exposé le processus de test basé sur l'utilisation d'un micro-testeur embarqué. Sous le contrôle du processeur, le micro-testeur teste les composants munis de wrappers IEEE 1500. Ce type de stratégie repose sur la réutilisation de quelques fonctionnalités du système pour effectuer le test de certains composants. Une partie du système est en mode fonctionnel (processeur embarqué, interconnect, micro-testeur etc.), pendant qu'une autre est en mode test (les coeurs testés). Ce comportement mixte fonctionnel/test au sein du même système impose certaines contraintes au niveau de la conception des coeurs testés.

Dans ce chapitre nous allons tout d'abord présenter l'architecture du wrapper IEEE 1500 ainsi que l'architecture interne du micro-testeur. Nous verrons comment la connexion entre les TPIUs et les wrappers est effectuée. Ceci nous amènera à présenter les contraintes de conception au niveau matériel sur les wrappers et sur les coeurs. Ensuite, les contraintes au niveau temporel (contraintes de timing) sur les signaux de test seront abordées.

Pour rester dans un domaine temporel, nous verrons pourquoi et comment le micro-testeur peut ajuster la fréquence de scan de chaque coeur. La manipulation des horloges de test nous permettra d'aborder le test *at-speed* des coeurs. En effet, avec les technologies actuelles, ce type de test est de plus en plus requis. Le micro-testeur effectue un test structurel des coeurs, nous verrons donc l'implémentation dans le micro-testeur des techniques de test *at-speed* structurels que sont le *Launch-On-Capture* et le *Launch-On-Last-Shift*.

9.1 Aspects matériels

9.1.1 Le wrapper de test au standard IEEE 1500

Le principal but de cette section est de présenter quelques détails architecturaux du wrapper, ainsi que le vocabulaire associé à la norme IEEE 1500. Ces informations proviennent de la documentation officielle du standard IEEE 1500 se trouvant sur le site [IEEE1500].

Le wrapper IEEE 1500 est composé :

- D'un ensemble de connecteurs formant l'interface série WSP (Wrapper Serial Port).
- D'un ensemble de connecteurs définis par l'utilisateur formant l'interface parallèle WPP (Wrapper Parallel Port).
- D'un registre d'instruction WIR (Wrapper Instruction Register)
- D'un registre permettant de contourner le wrapper WBY (Wrapper Bypass register).
- D'un registre de bordure WBR (Wrapper Boundary Register).

La figure 9.1 illustre les différents composants du wrapper IEEE 1500. Le wrapper possède différents modes de fonctionnement : les modes pour les opérations fonctionnelles, pour les opérations de test interne (INTEST), ou pour les opérations de test externe (EXTEST). D'autres modes permettent de déterminer si l'interface série est utilisée ou, l'interface parallèle si elle est présente.

Ces différents modes opératoires sont déterminés par l'instruction chargée dans le registre WIR. Une instruction est chargée dans le registre d'instruction du wrapper (WIR) grâce à la manipulation des signaux du WSP.

Les ports du wrapper

Le WSP est composé des signaux de contrôle Wrapper Serial Control (WSC) et d'un TAM d'une largeur de 1 bit Wrapper Serial Input (WSI) / Wrapper Serial Output (WSO). Le WSC est lui-même constitué de six signaux obligatoires :

- WRST : signal de reset du wrapper.
- WCLK : signal d'horloge du wrapper. Dans notre cas, ce signal d'horloge est branché à l'horloge du système.
- SelectWIR : signal permettant la sélection du registre WIR comme registre actif entre WSI et WSO.
- CaptureWR, ShiftWR, UpdateWR : signaux activant la capture, le décalage, et la mise à jour des données dans le registre sélectionné.

Le WPP est composé, de la même façon que le WSP, de signaux de contrôle Wrapper Parallel Control (WPC) et d'un TAM parallèle Wrapper Parallel Inputs (WPI) / Wrapper Parallel Outputs (WPO). Le WPP est optionnel et est défini par le concepteur du wrapper.

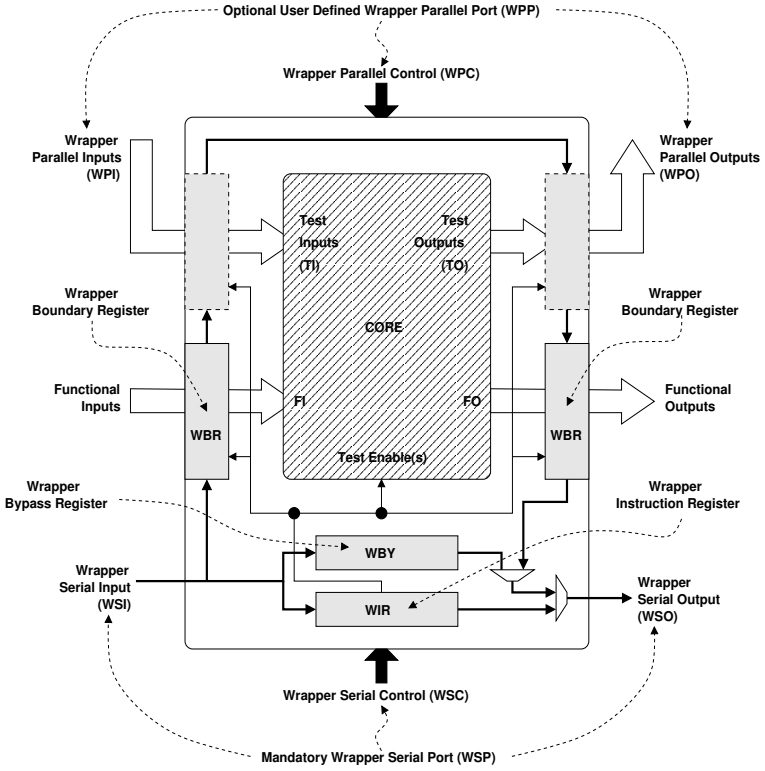


FIG. 9.1: Les différents composants d'un wrapper IEEE 1500.

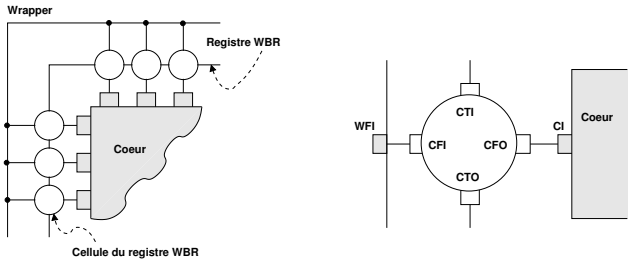
Les instructions du wrapper

Le standard prévoit un ensemble d'instructions définies pour utiliser uniquement l'interface série (WSP) et un ensemble d'instructions correspondant à l'interface parallèle (WPP). `WS_INTEST` est l'instruction permettant le test interne du coeur à travers l'interface série. L'instruction `WS_BYPASS` positionne le registre WBY comme registre actif entre WSI et WSO, permettant le contournement du wrapper. L'ensemble des instructions est défini dans la documentation officielle [IEEE1500].

Les cellules du registre WBR

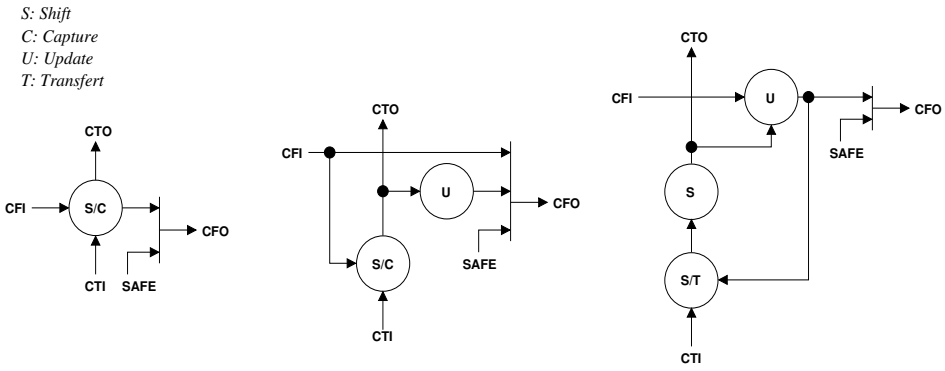
Le registre WBR est constitué d'un ensemble de cellules qui s'intercalent entre l'entrée du wrapper WFI (Wrapper Functionnal Input) et l'entrée du coeur CI (Core Input). La figure 9.2 présente le WBR ainsi que le détail des entrées/sorties d'une cellule du WBR. Le chemin CFI/CFO (Cell Functionnal Input/Cell Functionnal Output) constitue le chemin que prennent les données en mode fonctionnel. Le chemin CTI/CTO (Cell Test Input/Cell Test Output) correspond au chemin de test en mode décalage.

Différentes implémentations de cellules WBR sont prévues par la norme. La figure 9.3 présente le schéma conceptuel de différentes cellules WBR. Les symboles utilisés sont décrits dans l'annexe A du document [IEEE1500]. Ces cellules doivent gérer les évènements de capture des données (Capture),



(a) Le registre WBR *(b) Connectique d'une cellule du WBR*

FIG. 9.2: Le registre WBR et sa connectique.



(a) Cellule simple *(b) Cellule type IEEE 1149.1* *(b) Cellule complexe*

FIG. 9.3: Schémas conceptuels de différents types de cellules.

de mise à jour (Update), d'application des données aux entrées/sorties du coeur (Apply) et de transfert de données du chemin de test à la cellule la plus proche des entrées/sorties (Transfert). La valeur "SAFE" permet l'isolation du coeur. Lorsque le wrapper est en mode test interne, les valeurs "SAFE" sont envoyées vers l'extérieur et permettent de ne pas interférer avec la logique à l'extérieur du coeur. Lorsque le wrapper est mode test externe, les valeurs "SAFE" sont envoyées sur les entrées du coeur afin de protéger le coeur des effets du test externe. La présence de ces valeurs n'est pas obligatoire mais elle est recommandée par la norme.

9.1.2 Architecture interne du micro-testeur

Passons maintenant à l'étude de l'architecture interne du micro-testeur, et en particulier l'interface de celui-ci avec les wrappers IEEE 1500.

Comme nous l'avons présenté précédemment, l'architecture interne du micro-testeur est modulaire. Il est divisé en deux types de modules : le Prefetch-Buffer, sorte de mémoire cache partagée, et les TPIUs, modules exécutant les programmes de test HTC. La figure 9.4 rappelle cette architecture.

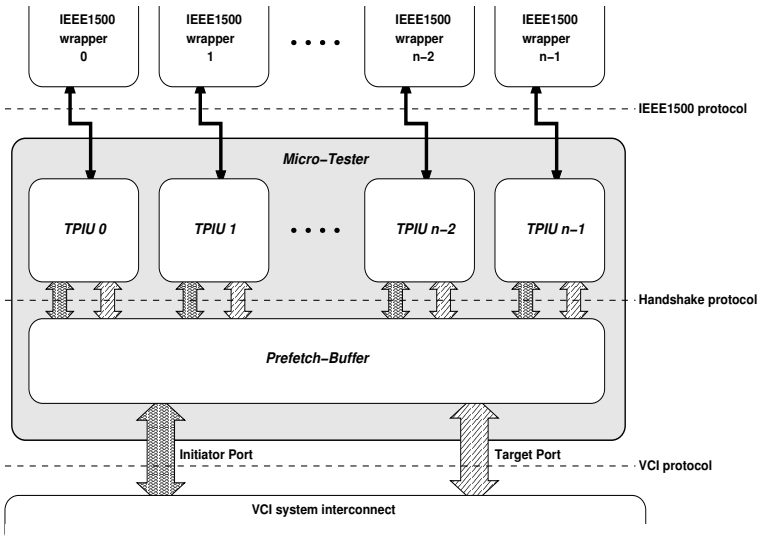


FIG. 9.4: Architecture modulaire du micro-testeur

Le Prefetch-Buffer

Le rôle défini pour le Prefetch-Buffer est de fournir une interface entre l'interconnect système et les TPIUs. Dans notre cas, le Prefetch-Buffer supporte le protocole VCI [VSIA]. Comme décrit précédemment, deux types de connexions sur l'interconnect système sont nécessaires : la première permettant la récupération des programmes de test, la deuxième permettant la communication avec le processeur. La figure 9.5 présente l'architecture interne du Prefetch-Buffer.

- **Port cible.** Un segment de l'espace d'adressage système est associé à ce port. Ainsi le processeur peut accéder (par lecture ou écriture) au micro-testeur. Ce port est contrôlé par l'automate *VCI Target*. Cet automate gère l'espace d'adressage complet du micro-testeur. Lorsque l'automate cible reçoit une adresse VCI, il décode l'adresse pour savoir avec quel TPIU le processeur désire communiquer. Les bits de poids fort correspondent à la cible micro-testeur, les bits de poids faible suivants correspondent au TPIU visé. Ainsi, l'automate avise le TPIU questionné, récupère les données et renvoie une réponse sur l'interconnect.
- **Port initiateur.** À travers ce port le PB agit comme un DMA en récupérant des fragments de programmes de test depuis la RAM externe du SoC. Le PB est une sorte de mémoire cache permettant de minimiser le temps d'oisiveté des TPIUs. Ainsi, il permet le test concurrent des coeurs. Les fragments des programmes de test sont stockés dans des mémoires FIFOs. Il y a une FIFO par TPIU permettant de stocker le programme de test correspondant au coeur auquel le TPIU est connecté (voir figure 9.5). La profondeur des FIFOs est un paramètre de l'architecture du PB, il peut prendre comme valeur : 4, 8, 16, 32 et 64. Afin de simplifier l'architecture, toutes les FIFOs ont la même profondeur. Les automates *VCI Initiateur Request* et *VCI Initiateur Response* se chargent des accès DMA. La sélection du TPIU à nourrir se fait par un algorithme *Round Robin* sur les FIFOs vides ou à moitié vides. Lorsqu'un TPIU a

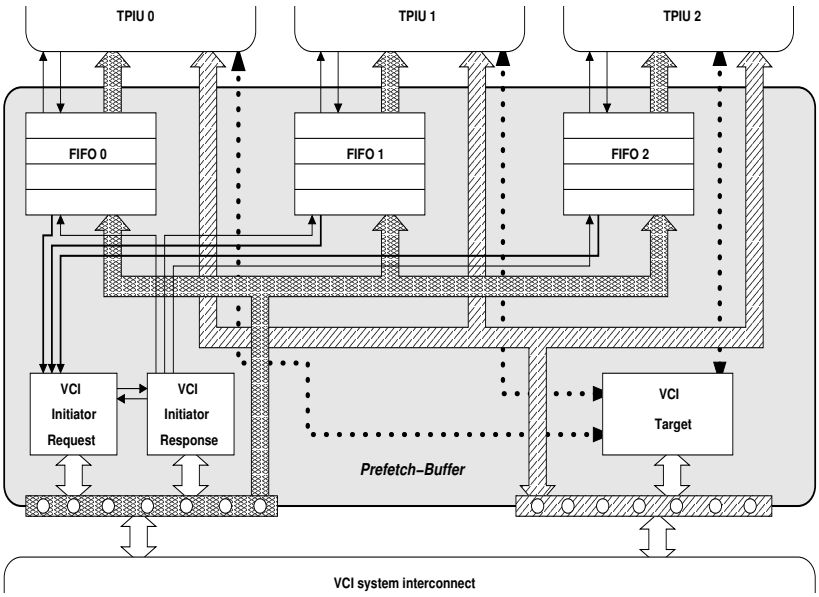


FIG. 9.5: Architecture interne du Prefetch-Buffer.

été sélectionné, une rafale de lecture dans la mémoire externe, à l'adresse du programme de test correspondant au TPIU sélectionné, est déclenchée. La taille de la rafale correspond à la moitié de la taille de la FIFO. La profondeur des FIFOs influence donc les temps de test.

Pour résumer, l'utilisation du Prefetch-Buffer permet de rendre les TPIUs complètement indépendants du protocole de communication utilisé par l'interconnect système. De plus, l'effet "cache" permet de minimiser l'oisiveté des TPIUs et de les faire fonctionner en parallèle.

L'interpréteur de programme de test (TPIU)

Le TPIU (Test Program Interpreter Unit) est l'unité permettant d'exécuter les programmes HTC et de les convertir en flux de données IEEE 1500. Il est architecturé autour de deux interfaces. D'un côté, il possède deux ports de communication avec le PB. Le premier port permet de récupérer les instructions HTC stockées dans la FIFO du Prefetch-Buffer, le deuxième, permet la configuration du TPIU. De l'autre côté, le TPIU est connecté au wrapper IEEE 1500. Comme le présente la figure 9.6 le TPIU est composé de deux sous-modules : le Configuration Manager et le Stream Manager.

Le Configuration Manager. Le TPIU possède un espace de configuration et de statut. Lorsque le processeur désire communiquer avec un TPIU, le Prefetch-Buffer établi alors un canal de communication entre le TPIU et le processeur. Le processeur peut alors communiquer avec la machine d'état du Configuration Manager, afin de lire ou d'écrire des données dans l'espace de configuration du

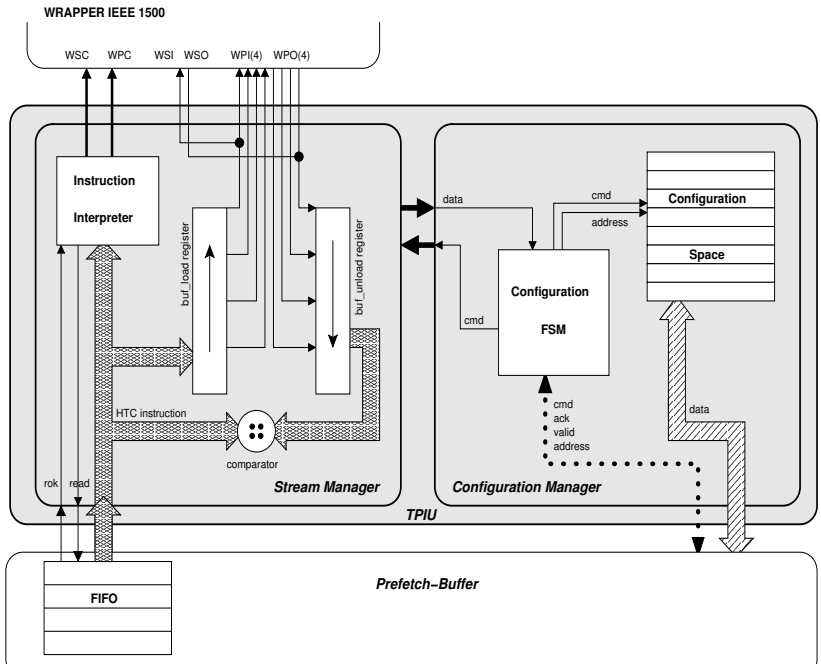


FIG. 9.6: Architecture interne du TPIU.

TPIU. Les informations que le processeur peut récupérer sont, par exemple, le statut du test : test en cours, le test a échoué, le test est réussi.

Le Stream Manager. Le Stream Manager est le module qui contient la machine d'état interprétant les instructions HTC. Dans cette architecture, chaque coeur possède un TAM ainsi que des signaux de contrôle de test privés. Comme le montre la figure 9.6, le Stream Manager est connecté au *Wrapper Serial Port* (WSP) et au *Wrapper Parallel Port* (WPP) [DaSilva03]. Ainsi, il fait office de contrôleur de test à travers le pilotage des signaux du *Wrapper Serial Control* (WSC) et du *Wrapper Parallel Control* (WPC). Les données de test sont envoyées et récupérées à travers le TAM série (*Wrapper Serial Input* WSI / *Wrapper Serial Output* WSO) ainsi qu'en utilisant l'interface parallèle (*Wrapper Parallel Inputs* WPI et *Wrapper Parallel Outputs* WPO). La figure 9.6 présente un TAM parallèle dont la largeur est de 4 bits.

Dans ce manuscrit nous nous sommes intéressés à la gestion des wrappers compatibles avec la norme IEEE 1500. Néanmoins, une modification légère de la machine d'état du Stream Manager permet de piloter n'importe quel protocole. Pour ce faire, il suffit de concevoir les instructions HTC correspondantes et de modifier le FSM en conséquence. Ainsi, en modifiant le FSM, le TPIU peut s'adapter au contrôle d'un port JTAG, d'un contrôleur BIST etc.

Le Stream Manager utilise un registre à décalage de 32 bits appelé Buf_Load, dont le but est de nourrir les entrées série et parallèle du wrapper. Le registre est connecté au ports WSI et WPI. Un deuxième registre à décalage de 32 bits, symétrique au premier, appelé Buf_Unload permet de rece-

voir les réponses de test (il est connecté aux ports WSO et WPO). Ce registre peut être configuré pour fonctionner comme un registre à décalage classique, ou, pour fonctionner comme un MISR (Multiple Inputs Signature Register). Lorsque ce registre est configuré en registre à décalage classique, la réponse donnée par le coeur est comparée bit à bit avec celle attendue. Ainsi il est possible de déterminer le bit fautif, permettant d'effectuer un diagnostic précis du coeur. Lorsque ce registre est configuré en mode MISR, la signature sur 32 bits obtenue après le test complet du coeur est suffisante pour déterminer s'il est fautif ou non. Le mode compaction (registre Buf_Unload configuré en mode MISR) permet donc une réduction drastique de la taille des programmes HTC, puisque toutes les réponses attendues sont réduites à une unique signature sur 32 bits.

Interface parallèle. L'entrée parallèle du wrapper (WPI/WPO) est alimentée par le registre à décalage Buf_Load. Les signaux WPI sont tous connectés sur ce registre, et sont espacés de façon régulière. Le registre, qui est sur 32 bits, est en quelque sorte divisé en autant de sous registres à décalage que la largeur du TAM. Par exemple, si la largeur du TAM est de 2, les 16 bits de poids faible alimente WPI(0), les 16 bits de poids fort alimente WPI(1). En effet, WPI(0) est branché au bit 0 du registre Buf_Load, WPI(1) est branché au bit 16. L'algorithme permettant de brancher WPI au registre Buf_Load est présenté dans la figure 9.7 (a). Lorsque la largeur de TAM n'est pas un diviseur de 32, les bits de poids fort (reste de la division entière de 32 par la largeur du TAM) du registre Buf_Load sont alors inutilisés. Dans les programmes HTC des bits de bourrage doivent donc être insérés aux emplacements de ces bits inutilisés. La figure 9.7 (b) présente les branchements du WPI lorsque la largeur du TAM est 3. Cette largeur n'étant pas un diviseur de 32, les deux bits de poids forts sont donc des bits de bourrage.

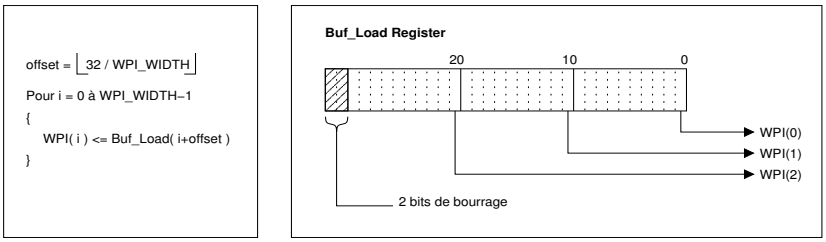


FIG. 9.7: Gestion de l'interface parallèle (a) et exemple d'un TAM de largeur 3 (b)

Le pire cas est obtenu lorsque la largeur du TAM est de 17, on a alors 15 bits de bourrage. **Notre architecture est donc sensible à la largeur du TAM**. En effet, les bits de bourrage enflent la taille des programmes HTC et augmentent le temps de test. De plus, cette architecture fixe de 32 bits à pour conséquence de restreindre la largeur maximale du TAM parallèle géré à une largeur d'au maximum 32 bits.

9.1.3 Conséquences architecturales sur les wrappers et les coeurs

Le principal but de cette section est de présenter l'impact, au niveau conception, sur les wrappers et sur les coeurs, de l'utilisation de notre micro-testeur comme TAM. Dans un premier temps, nous

verrons comment gérer le flux discontinu des données de test injectées dans les wrappers par le micro-testeur. Dans un deuxième temps, nous évoquerons l'effet "Ripple" des wrappers et comment il doit être géré.

Gestion du flux discontinu des données test

Le micro-testeur injecte les stimuli de test et récupère les réponses par paquet de 32 bits (dû au registre de 32 bits Buf_Load et Buf_Unload, cf section 9.1.2). Entre deux paquets, il y a au moins un cycle d'attente, le temps de charger le registre Buf_Load. Ce nombre de cycles peut être plus grand si la FIFO contenant le programme de test HTC est vide. Ainsi, les données test sont envoyées et récupérées par intermittence, de façon discontinue.

Notre stratégie suppose que les horloges de test des wrappers et des coeurs sont branchées aux horloges systèmes. Donc, pendant les chargements/déchargements des chaînes de scan, des cycles d'attente se produisent. Or, les chaînes de scan classiques supposent que le flux de données est envoyé de façon continue. En effet, lorsque le chemin de test est implémenté avec des bascules scan décrites par la figure 9.8 (a), seulement deux modes sont disponibles : décalage ou fonctionnel. Lorsque le décalage est désactivé (signal scan_enable à 0), les cycles d'attente font basculer la cellule en mode fonctionnel (perdant ainsi les valeurs chargées jusque là).

Afin de pouvoir supporter un flux discontinu de données test, deux stratégies s'offrent à nous. La première est de définir une nouvelle bascule de test, gérant les cycles d'attente. la deuxième est de mettre en place un contrôle de l'horloge du wrapper et du coeur.

De nouvelles bascules de test. Comme nous venons de le préciser, les bascules scan standards disposent de deux modes de fonctionnements (figure 9.8 (a)) :

- Fonctionnel : signal `SCAN_ENABLE` à 0, la bascule prend en entrée la valeur `FUNC`.
- Décalage (scan) : signal `SCAN_ENABLE` est actif (1), la bascule prend en entrée la valeur `SCAN_IN`.

Pour répondre aux contraintes imposées par l'envoi et la réception discontinus du flux de données test, une nouvelle bascule scan a été développée. Par rapport aux bascules scan standard, les nouvelles bascules de test doivent disposer d'un mode permettant de garder leur valeur précédente, lorsque le flux de données de test est interrompu. Ainsi ces nouvelles bascules doivent disposer d'au moins trois modes (figure 9.8 (b)) :

- Fonctionnel : dans ce mode la bascule prend en entrée la valeur `FUNC`.
- Décalage (scan) : la bascule agit comme une bascule scan standard prenant en entrée la valeur `SCAN_IN`.
- Hold : la bascule garde sa valeur précédente.

Afin d'obtenir trois modes différents pour la bascule, deux signaux de commande sont nécessaires : `TEST_MODE(1)` et `TEST_MODE(0)`. Le tableau 9.1 résume l'utilisation d'un telle bascule.

On peut voir sur le schéma (b) de la figure 9.8 que le multiplexeur ajouté n'est pas sur le chemin critique du mode fonctionnel (le signal `FUNC`). Les performances temporelles en mode fonctionnel sont

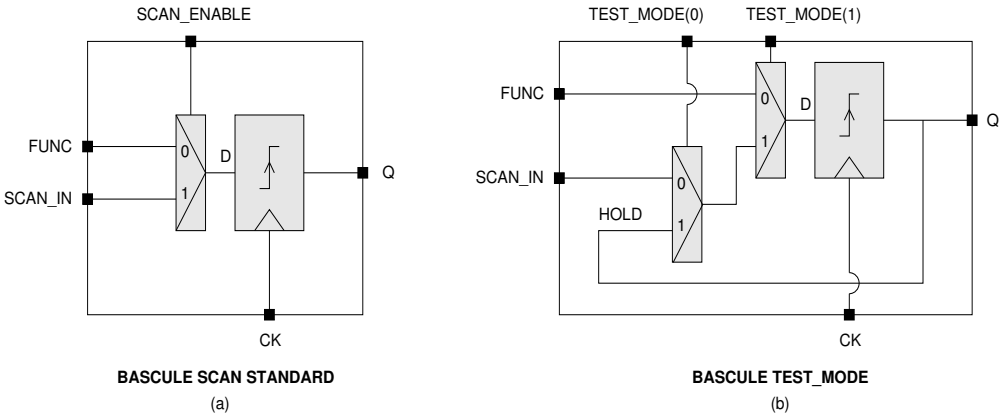


FIG. 9.8: Bascule de test classique (a) et nouvelle bascule de test (b)

TEST_MODE		Mode	Commentaire
(1)	(0)		
1	*	FUNC	Le circuit est en mode fonctionnel
0	1	SCAN	La chaîne de scan est décalée
0	0	HOLD	La chaîne de scan garde son ancienne valeur

TAB. 9.1: Fonctionnement de la nouvelle bascule de test.

donc identiques à la version standard. En revanche, l'ajout d'un nouveau multiplexeur augmente la surface de la bascule. Afin d'évaluer le coût en surface de ces nouvelles bascules, nous avons modifié l'outil SCAPIN (Scan-Path Insertion), afin qu'il puisse insérer dans une netlist de portes logiques ce nouveau type de bascules scan. SCAPIN, outil de la chaîne de CAO ALLIANCE [ALL], fonctionne sur une netlist de portes logiques de la bibliothèque de cellules standards SXLIB. L'augmentation de la surface peut être observée au niveau de la bascule elle-même, ainsi qu'au niveau du circuit.

- **Niveau bascule.** L'insertion d'un chemin de test classique (bascule de type (a)) ajoute **33%** de surface supplémentaire à chaque bascule. L'insertion d'un chemin de test avec des bascules de type (b) augmente chaque bascule de **55%**. On peut noter que l'augmentation de la surface due à l'utilisation d'une bascule de type (b) par rapport à une bascule scan classique (de type (a)) est de **16%**.
- **Niveau circuit.** Une analyse de l'augmentation de surface a été effectuée sur un petit circuit d'environ 1360 cellules SXLIB. Ce circuit contient 150 bascules. Le circuit contenant un chemin de test classique ajoute **9.5%** de surface additionnelle. Le circuit contenant un chemin de test utilisant les nouvelles bascules augmente de **6%** cette surface. L'augmentation par rapport au circuit sans chemin de test est de 16%. Les résultats sont donnés pour des circuits placés/routés.

Si le coeur contient ce nouveau type de bascule, et que les signaux "scan_enable" du coeur sont gérés par le wrapper, l'interface entre le wrapper et le coeur doit être modifiée afin que le wrapper puisse

contrôler les deux signaux "TEST_MODE".

De la logique sur l'horloge. Une autre façon de rendre les cycles d'attente non visibles est d'inhiber l'horloge du wrapper et du coeur pendant ces cycles. De la logique sur l'horloge du coeur et du wrapper permettent de masquer ces cycles. Faire de la logique sur l'horloge est fréquemment utilisé dans le monde du test afin de générer une horloge de test différente de l'horloge fonctionnelle. Ainsi, l'interface du TPIU doit être modifiée afin de gérer un signal *clk_enable* permettant de masquer l'horloge du coeur et du wrapper (figure 9.9).

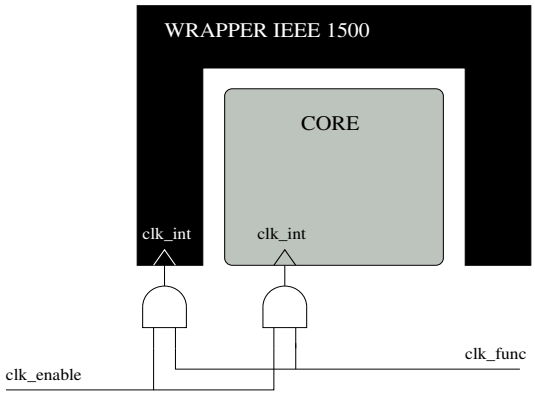


FIG. 9.9: Logique sur l'horloge du coeur et de son wrapper.

Les micro-testeurs et l'effet "Ripple"

Nous avons vu dans la section 9.1.1, concernant le wrapper IEEE 1500, que les cellules du registre de bordure WBR pouvaient être implémentées de plusieurs façons (figure 9.3). En particulier, nous avons porté notre attention sur la présence des valeurs "SAFE", qui permettent l'isolation du coeur. Nous avons vu qu'en mode test interne, ces valeurs sont envoyées vers l'extérieur afin que le test du coeur n'interfère pas avec le monde extérieur. En effet sans la présence de ces valeurs "SAFE", les valeurs des vecteurs en cours de chargement/déchargement sont visibles sur l'interface extérieur du coeur ("ripple-while-shift"). La présence de ces valeurs "SAFE" n'est pas obligatoire, mais est recommandée par la norme.

Cependant, dans le cas d'un test exécuté par un micro-testeur (quel qu'il soit), si le coeur est connecté au même interconnect que le micro-testeur et/ou le processeur, le wrapper doit positionner obligatoirement ces valeurs "SAFE" sur son interface externe. En effet, si ces valeurs ne sont pas présentes, lorsque des vecteurs sont chargés ou déchargés, ils peuvent, en étant visibles sur l'interconnect système, interférer avec les transactions en cours. Il est fort probable, que le chargement/déchargement des vecteurs de test dans le WBR exécutent une requête sur l'interconnect, paralysant les communications en cours.

Pour conclure, cette étude architecturale du micro-testeur nous a permis de voir les retombées sur la conception du wrapper et du coeur testé. Nous avons vu comment gérer l'effet "Ripple" et le flux discontinu des données test par le biais d'une nouvelle bascule de test ou par le contrôle de l'horloge. Voyons maintenant quelles sont les contraintes au niveau temporel.

9.2 Aspects temporels

Dans cette section nous verrons les aspects temporels liés à l'utilisation du micro-testeur. Dans un premier temps nous examinerons les différentes contraintes sur la fréquence de décalage (scan) dans un circuit. Nous verrons comment notre architecture permet de sélectionner une fréquence de test associée pour chaque coeur. Ensuite nous présenterons les contraintes temporelles sur les signaux de commande émis par le micro-testeur, dans le cas d'un coeur muni de bascules "TEST_MODE", puis dans le cas ou le micro-testeur contrôle l'horloge.

9.2.1 Fréquence d'horloge en mode test

Il y a deux principales sources d'horloge de test. La première est celle fournie par l'ATE externe, la deuxième est celle embarquée, fournie par la puce. Traditionnellement, le testeur externe a toujours alimenté les horloges de test. Cependant, la sophistication et le coût des testeurs externes croissent avec la fréquence d'horloge et la précision nécessaire. La deuxième source d'horloge peut provenir de l'intérieur de la puce elle-même. De plus en plus de circuits contiennent une PLL (phase-locked loop) ou une autre circuiterie embarquée génératrice d'horloge. Utiliser ces horloges fonctionnelles pour le test peut apporter plusieurs avantages par rapport à l'utilisation de l'horloge de l'ATE. La plus remarquable étant que l'utilisation des horloges haute vitesse embarquées réduisent les exigences des ATEs, permettant l'utilisation de testeurs moins sophistiqués, donc moins onéreux.

Le choix du micro-testeur embarqué a été fait dans l'optique d'utiliser l'horloge interne fonctionnelle comme horloge de test.

La fréquence de décalage (scan), généralement plus lente que la fréquence nominale du circuit, influence considérablement le temps de test. En effet, pour 1 cycle de capture d'une réponse à un vecteur donné, N cycles ont été nécessaires pour le charger, et encore N cycles sont nécessaires pour décharger la réponse. Les deux facteurs principaux limitant, dans un test classique, la vitesse de décalage sont : les capacités du circuit et les capacités du testeur externe.

Les capacités du circuit

Une première raison de ralentir la fréquence de décalage par rapport à la fréquence du circuit est due au routage du signal scan-enable. Ce signal est global puisqu'il attaque toutes les bascules scan du circuit. Pourtant, ce signal n'est en général pas "bufferisé" correctement pour éviter d'ajouter de la surface supplémentaire liée au test. Ainsi, ce signal peut ne pas atteindre les exigences temporelles de la fréquence nominale du circuit.

Une deuxième raison est liée à la consommation du circuit en mode test. La consommation dépend de la commutation des noeuds du circuit et de la fréquence d'application du test [Girard02]. Alors qu'en mode fonctionnel 15 à 20% des noeuds du circuit commutent, lors du test, cette activité passe à 35-40%. Un moyen de réduire la consommation due à une commutation importante des noeuds, est de réduire la fréquence d'application du test.

Les capacités du testeur externe

Comme nous l'avons précisé, une horloge rapide augmente le coût du testeur. Les testeurs bas coût ne fournissent généralement pas de telles horloges.

De plus, [Beck05] nous prévient que même si le testeur fournit une horloge rapide, il est probable que cette horloge ne puisse être transmise au circuit. En effet les SoCs actuels, et particulièrement les micro-contrôleurs, communiquent avec leur environnement à une fréquence très faible. Les plots du composant sont lents comparés à la fréquence de fonctionnement interne. L'application d'une horloge externe rapide nécessiterait donc l'ajout d'un plot rapide capable de transmettre le signal d'horloge.

Dans les processeurs PXA25x, PXA26x, et PXA27x Intel Xscale, le test peut s'effectuer a une fréquence de 100 à 400 MHz en utilisant une circuiterie interne, alors que la fréquence de décalage fournie par le testeur externe n'est que de 13MHz [Press06]. Dans [Tendolkar00] les techniques de conception pour le test du PowerPC MPC7400 permettent d'effectuer un test *at-speed* à 720MHz, et d'effectuer le décalage des vecteurs jusqu'à 80MHz. On voit qu'effectivement, un test généré en interne permet d'utiliser des fréquences assez élevées.

Bien que les contraintes liées aux capacités du circuit ne permettent pas toujours de pouvoir décaler les vecteurs de test à la vitesse nominale du circuit, le facteur principal de ralentissement des horloges de test, est donc la difficulté d'application d'horloge rapide depuis l'extérieur.

Sélection de la fréquence de test dans le micro-testeur

L'avantage de l'utilisation du micro-testeur est que l'on peut s'affranchir du testeur externe. Les fréquences de test ne dépendent que des capacités du circuit. Néanmoins, il reste que le micro-testeur doit adapter sa fréquence de test en fonction des capacités du circuit.

Chaque TPIU dispose d'un registre compteur, permettant d'augmenter ou de réduire le nombre de cycles entre deux cycles de scan. Une instruction HTC permet de régler de façon logicielle la valeur de ce compteur (SHIFT_FREQ voir tableau D.1). Une fois cette fréquence réglée, lorsque le FSM du *Stream Manager* décode une instruction HTC correspondant à l'application d'un vecteur de test, le parcours des états est celui présenté en figure 9.10. L'automate présenté décrit l'application d'un vecteur, du chargement à la capture de la réponse.

L'automate est dans l'état *IDLE*, lorsqu'il rencontre l'instruction HTC d'envoi de vecteur, il passe dans la succession d'état *WAIT*, *SHIFT WAITLS* et *CAP*, pour finalement revenir dans l'état *IDLE*. L'état *SHIFT* effectue un cycle de scan, l'état *WAIT* permet de temporiser entre deux cycles de scan. Tant que le registre compteur n'est pas arrivé à 0, on reste dans cet état. Ainsi, le basculement entre ces

deux états permet de gérer la fréquence du test. L'état *WAITLS* (Wait Last Shift) à la même fonction que l'état *WAIT*, permettant de temporiser avant la capture des réponses (état *CAP*).

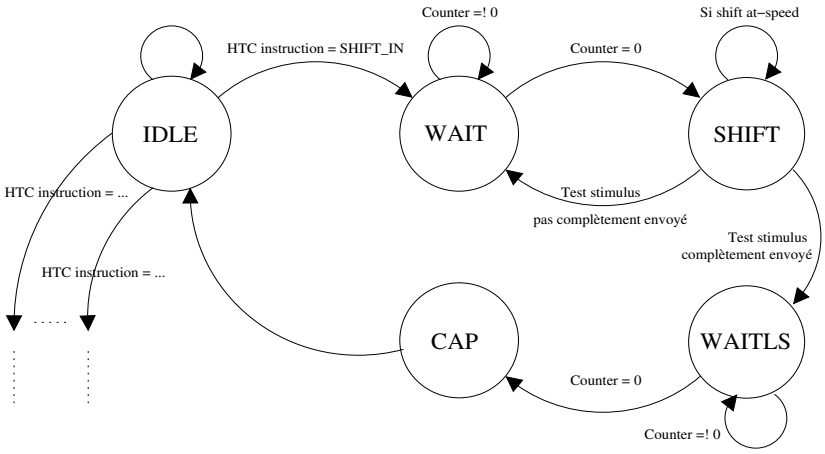


FIG. 9.10: Branche de l'automate du TPIU permettant l'application d'un pattern de test.

La fréquence de test (f_{scan}) dépend de la fréquence du système (f_{sys}) : $f_{scan} = f_{sys}/x$ où x est un entier $\in [1; 32]$ (le registre "Counter" est sur 5 bits). La fréquence de test peut varier de $f_{scan} = f_{sys}/32$, la plus lente, à $f_{scan} = f_{sys}$ la plus rapide (scan *at-speed*). Si l'on suppose une fréquence système de 300 MHz, la fréquence de test peut varier de 300 MHz à environ 9 MHz, en passant par des valeurs telles 150, 100, 75 ou bien 50 MHz.

9.2.2 Contraintes temporelles sur les signaux de commande

Le chargement/déchargement d'un vecteur à une certaine fréquence f_{scan}, impose que le micro-testeur fonctionnant à la fréquence f_{sys}, envoie les signaux de commande sous certaines contraintes de temps. Afin de mieux comprendre ces contraintes temporelles, nous allons reprendre l'exemple précédemment décrit, d'application d'un vecteur de test, présenté en figure 9.10. Etudions d'abord ces contraintes lorsque le coeur est muni de bascules "TEST_MODE", puis lorsque de la logique sur l'horloge est utilisée.

Pour un coeur muni de bascules TEST_MODE

La figure 9.11 montre le chronogramme associé à l'exécution de cette séquence pour seulement deux décalages et la capture de la réponse. Le circuit utilise des bascules TEST_MODE et la fréquence de scan $f_{scan} = f_{sys}/4$.

A chaque fois que l'automate est dans l'état *SHIFT* le signal ShiftWR du WSC est activé, ce qui provoque de façon combinatoire le passage des bascules TEST_MODE en mode SCAN. Dès que le signal ShiftWR est désactivé, les bascules TEST_MODE repassent en mode HOLD. Lorsque le signal CaptureDR du WSC passe à la valeur 1, les bascules TEST_MODE passent dans l'état fonc-

tionnel (FUNC) afin de capturer la réponse du circuit. Tout de suite après, les bascules TEST_MODE repassent en mode HOLD afin de garder l'information.

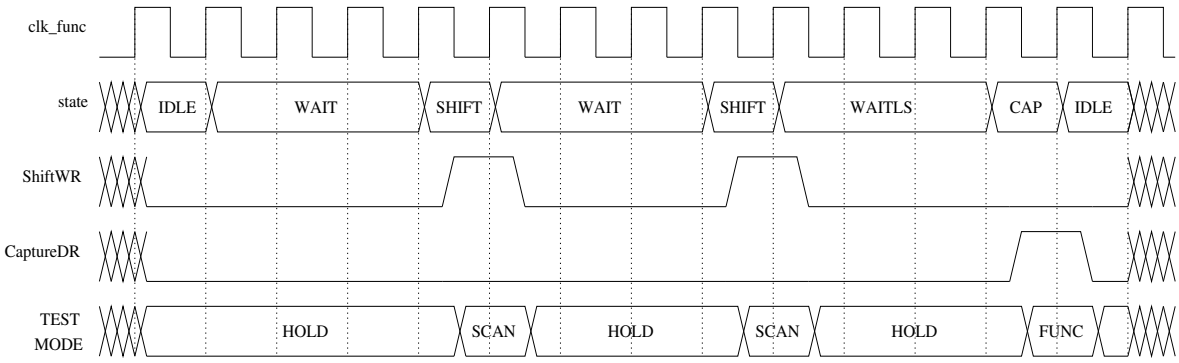


FIG. 9.11: Application d'un vecteur en présence de bascules TEST_MODE.

La figure 9.12 effectue un zoom sur les contraintes de temps liées à l'utilisation de ces bascules. Les contraintes sont que les signaux TEST_MODE doivent respecter les temps de *setup (Ts)* avant le front montant, et les temps de *hold (Th)* après le front montant (ce qui est vrai pour n'importe quel signal). De même, le signal ShiftWR (et CaptureDR) doit respecter ses temps *Ts* et *Th* afin de permettre au signal TEST_MODE de se stabiliser.

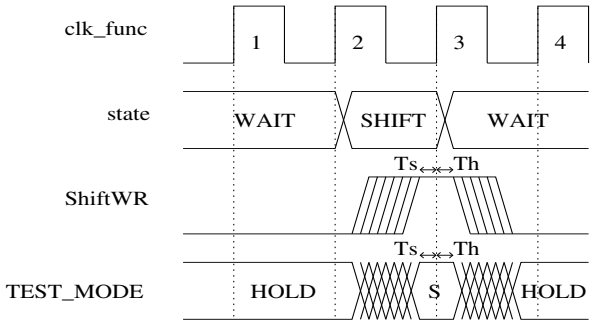


FIG. 9.12: Ts et Th de différent signaux.

Ces signaux ont, un cycle pour s'activer, un cycle pour se désactiver, comme n'importe quel signal du circuit. Ils doivent respecter les temps *Ts* et *Th*. Le seul point critique est que les signaux TEST_MODE, qui sont connectés à toutes les bascules du coeur, dépendent du WIR et des signaux ShiftWR et CaptureDR (de façon combinatoire) émis par le micro-testeur, créant ainsi **une chaîne longue potentielle**.

Pour un coeur ayant de la logique sur l'horloge

Comme précédemment, nous reprenons l'exemple de la figure 9.10. La figure 9.9 rappelle la logique de contrôle de l'horloge (à base de portes "AND").

La figure 9.13 montre le chronogramme associé à l'exécution de cette séquence pour seulement deux décalages et la capture de la réponse. Le circuit utilise de la logique sur l'horloge au lieu des bascules TEST_MODE.

A chaque fois que l'automate est dans l'état *SHIFT*, le signal clk_enable est activé afin d'autoriser l'application d'un cycle d'horloge interne clk_int. Le signal scan_enable (interne au coeur) est conditionné par la valeur du signal ShiftWR (scan_enable prend la valeur de ShiftWR). Ce dernier signal est laissé à la valeur 1 pendant toute la durée du décalage du vecteur (même pendant l'état *WAIT* à la différence de la gestion des bascules TEST_MODE) afin de pouvoir gérer des signaux scan_enable "lent". Le signal CaptureDR n'est pas représenté, mais il reste identique à celui présenté dans le chronogramme en figure 9.11.

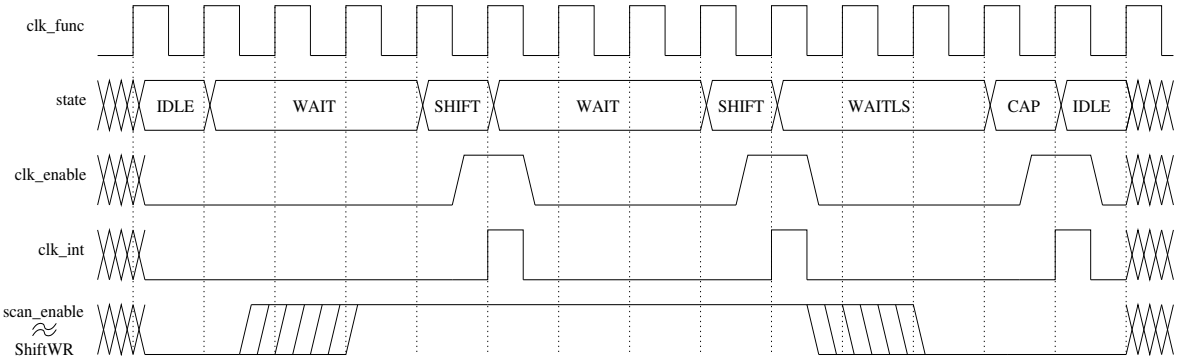


FIG. 9.13: Application d'un vecteur en présence de logique sur l'horloge.

Le signal clk_enable est sensible au timing. En effet le chronogramme sur la figure 9.14 nous montre qu'il doit être correctement positionné afin de générer un cycle correct sur l'horloge interne clk_int. Pour ceci, trois contraintes temporelles doivent être adressées :

- *Th0* : c'est la durée pendant laquelle le signal clk_enable doit rester à la valeur 0 après le front montant 2 (lorsque l'automate est passé dans l'état SHIFT). Th0 doit durer une demie période de cycle fonctionnel, du front montant du cycle 2 au front descendant du même cycle.
- *Ts* : c'est le temps de stabilisation du signal à la valeur 1 avant le front montant du cycle 3.
- *Th1* : c'est la durée pendant laquelle le signal clk_enable doit rester à 1 après le front montant 3. Th1 doit durer une demie période de cycle fonctionnel, du front montant du cycle 3 au front descendant du même cycle.

Le chronogramme de la figure 9.15 nous montre l'effet d'un *Th0* inférieur à une demie période d'horloge fonctionnelle. Dans ce cas, le signal clk_enable arrive prématurément et deux fronts montants sur clk_int sont générés. Une bonne façon de respecter ces contraintes est d'utiliser, dans le TPIU, une bascule sur front descendant pour générer le signal clk_enable.

Bien que cette méthode de contrôle de l'horloge fonctionnelle impose que le signal clk_enable s'active et se désactive en **une demie période d'horloge fonctionnelle**, les signaux tels que le scan_enable ou les signaux du WSP (ShiftWR ou CaptureDR par exemple) **peuvent alors être des signaux "lents"**.

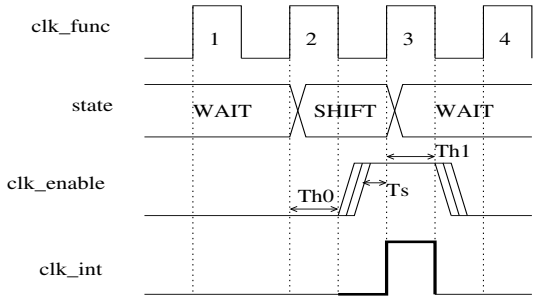


FIG. 9.14: Signal clk_enable correct.

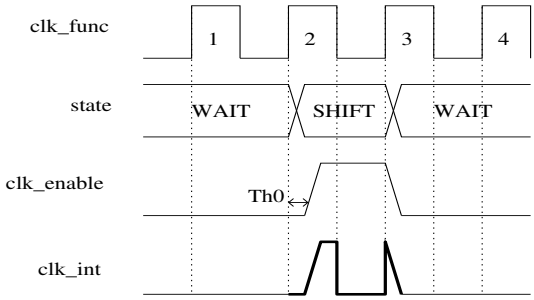


FIG. 9.15: Signal clk_enable prématuré.

9.3 Le test At-Speed / AC-Scan

Pour rester dans un domaine temporel, cette section décrit l'implantation du test *at-speed* dans le micro-testeur.

Pendant de nombreuses années le test *at-speed* a été l'apanage des circuits à haute performance tels que les microprocesseurs "full-custom" [Carbine97, McLaurin00]. Le principal intérêt était de s'assurer qu'une vitesse spécifique pouvait être atteinte. Ce type de test dit "speed binning" a été le précurseur du test *at-speed*. Cependant, avec les technologies récentes, des défauts comme les vias partiellement remplis ou les "mouse-bites" sur les lignes d'interconnexions sont de plus en plus présents. De plus, de nouveaux problèmes de conception tels que la diaphonie (cross-talk) ou les chutes de tension (voltage drop) ont des effets non négligeables sur la vitesse de transition. Enfin, les phénomènes de vieillissement peuvent aussi affecter les caractéristiques d'un transistor. Ceci peut résulter dans un comportement incorrect du transistor affecté, en général, il devient "plus lent" [Bennetts06]. Ces défauts, affectant le circuit d'un point de vue temporel, peuvent ne pas être détectés par des tests statiques, effectués à faible fréquence (tests DC-scan). C'est pourquoi, des tests dynamiques, où les réponses sont capturées à la fréquence nominale du circuit (tests AC-scan), s'assurant que le circuit fonctionne à sa vitesse nominale, sont devenus une nécessité pour tous les types de circuits.

Cette section est découpée comme suit. Tout d'abord, nous verrons quels sont les modèles de fautes considérés. Ensuite, nous décrirons les techniques de test *at-speed* utilisant les chaînes de scan (test AC-scan). Finalement, nous verrons comment nous avons introduit ces techniques dans le micro-testeur.

9.3.1 Les modèles de fautes

Afin d'être manipulés par les outils de génération de vecteurs de test, ces défauts sont représentés par des modèles appelés fautes temporelles ou fautes de délai.

Pour détecter une faute temporelle, il faut provoquer une transition dans un chemin combinatoire et vérifier que la valeur attendue sur la sortie a été propagée dans les délais spécifiés. Ainsi, l'application d'un test *at-speed* nécessite l'application de deux vecteurs. Le premier vecteur sensibilise le chemin combinatoire, le second vecteur lance la transition et, la capture de la réponse doit se faire à la vitesse

d'horloge du système. Si la réponse capturée indique que la logique impliquée n'a pas effectué la transition attendue dans le temps de cycle, le chemin a échoué au test et est considéré comme contenant un défaut.

Deux modèles de fautes sont principalement utilisés aujourd'hui : un modèle local (faute de transition ou transition-delay fault [Koeppe86, Abraham85]) et un modèle global (faute de chemin ou path-delay fault [Smith85]).

Faute de transition

Une faute de transition provient d'un défaut local, affectant le temps de transition d'un signal. Cette transition n'atteint plus le point d'observation, le long d'un chemin sensibilisé, en une période (d'horloge) requise. Il existe deux types de fautes de transition possibles : la transition "lente-à-monter" (slow-to-rise) et la transition "lente-à-descendre" (slow-to-fall) [Landrault96]. La figure 9.16 expose la localité d'une faute de transition.

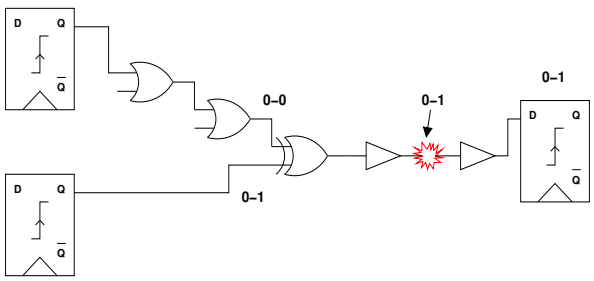


FIG. 9.16: Faute temporelle de transition.

Faute de chemin

Une faute de chemin provient d'un défaut global, responsable d'une accumulation de retards d'un chemin combinatoire donné qui dépasse une durée spécifiée. Pour chaque chemin combinatoire dans un circuit, il y a deux types de fautes : une transition montante, une descendante. La figure 9.17 montre une faute temporelle sur un chemin.

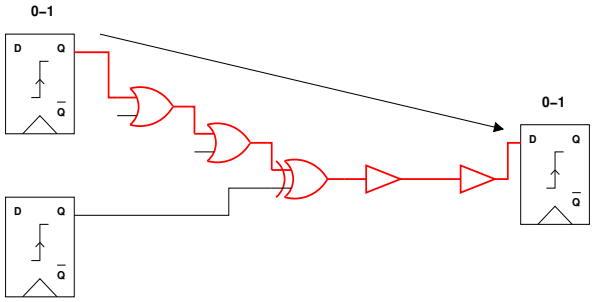


FIG. 9.17: Faute temporelle sur un chemin.

La sélection des chemins à tester par cette méthode se fait par analyse temporelle statique (static timing analysis). Le choix se porte sur les chemins critiques (chaînes longues), en effet ces chemins sont plus sensibles que les autres au timing.

9.3.2 Les techniques d'application

Pour appliquer un test *at-speed* structurel, les chaînes de scan sont utilisées. Dans un environnement de chaînes de scan classiques, deux techniques différentes d'application de ces test à deux vecteurs ont vu le jour : le *Launch-On-Last-Shift* (LOLS) et le *Launch-On-Capture* (LOC) [Savir92, Patil92, Savir94]. Le premier vecteur V1 initialise le circuit. Le second vecteur V2 est utilisé pour lancer les transitions. Les propagations de ces transitions sont capturées en retour dans les chaînes de scan. Si le second vecteur V2 est obtenu par décalage d'un seul bit du premier vecteur V1, cette technique est appelée *Launch-On-Last-Shift*. Si le second vecteur V2 est obtenu par lancement d'un cycle fonctionnel du coeur, cette technique est appelée *Launch-On-Capture*.

Ces deux techniques imposent des contraintes temporelles différentes sur le signal "scan_enable". Etudions le LOLS d'abord, puis le LOC.

Launch-On-Last-Shift

La figure 9.18 présente le chronogramme illustrant le timing associé à cette technique. Dans ce chronogramme, lorsque le signal `scan_enable` est bas (0) le circuit opère en mode fonctionnel, lorsqu'il est haut (1) le circuit est en mode décalage (scan). Donc, le signal `scan_enable` doit rester à l'état haut pendant toute la durée du décalage du vecteur V1. Toujours dans cette figure, les bascules sont supposées fonctionner sur front montant de l'horloge. Comme nous l'avons dit, dans la technique du LOLS, le second vecteur V2 est obtenu par décalage d'un bit du vecteur V1. Dans ce cas, le signal `scan_enable` doit rester haut (en mode décalage) un cycle après le chargement complet du vecteur V1. Lorsque V2 a été généré, le signal `scan_enable` doit alors être rapidement désactivé, en un cycle d'horloge fonctionnelle, afin que les réponses du circuit au vecteur V2, soient alors capturées dans les bascules.

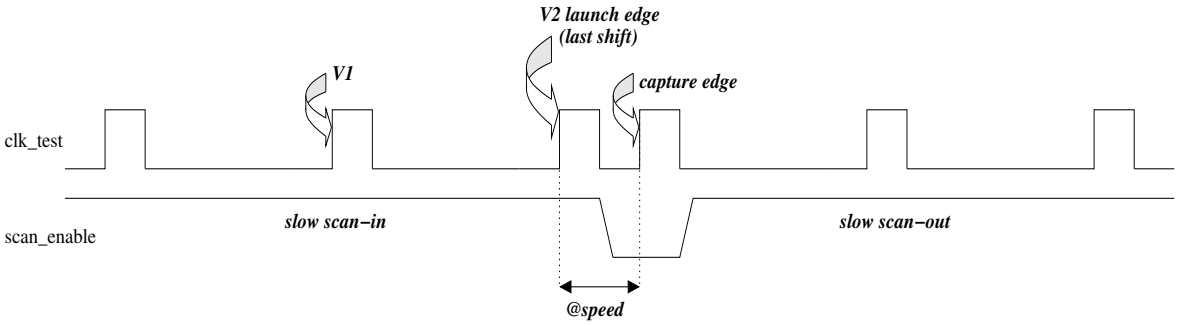


FIG. 9.18: Chronogramme de la technique dite "Launch-On-Last-Shift".

Le signal scan-enable est un signal global puisqu'il attaque toutes les bascules scan du circuit. Or

dans la technique LOLS, ce signal doit pouvoir commuter en un cycle d'horloge fonctionnelle. Pour ce faire, il doit être routé avec des contraintes de temps sévères [Nadeau-Dostie00].

Launch-On-Capture

La figure 9.19 présente le chronogramme qui illustre le timing associé à la technique du *Launch-On-Capture*. Comme pour la précédente figure, lorsque le signal `scan_enable` est à 0, le circuit opère en mode fonctionnel, à 1 en mode décalage. Dans le cas du LOC, le signal `scan_enable` doit rester haut (en mode décalage) pendant le chargement complet du vecteur V1. Ensuite, le signal `scan_enable` est désactivé (état bas). On peut alors laisser s'écouler assez de temps pour que le changement d'état sur ce signal (lent, parce que global) puisse prendre effet à travers tout le circuit. Ensuite, deux cycles d'horloge à la vitesse du système peuvent s'appliquer pour lancer V2 puis pour capturer les réponses du circuit sous test. Parce que le signal `scan_enable` est bas (mode fonctionnel) au premier coup d'horloge système, le vecteur V2, qui est la réponse du circuit au vecteur V1, est capturé dans les bascules ("V2 launch edge" sur la figure 9.19). Le second coup d'horloge système capture la réponse de V2 dans les bascules ("capture edge" sur la figure 9.19). Le temps entre les deux cycles d'horloge fonctionnelle est la durée d'un cycle à la fréquence nominale (*at-speed*). Ensuite les réponses capturées dans les bascules peuvent être récupérées par décalage de la chaîne de scan à une fréquence moins élevée.

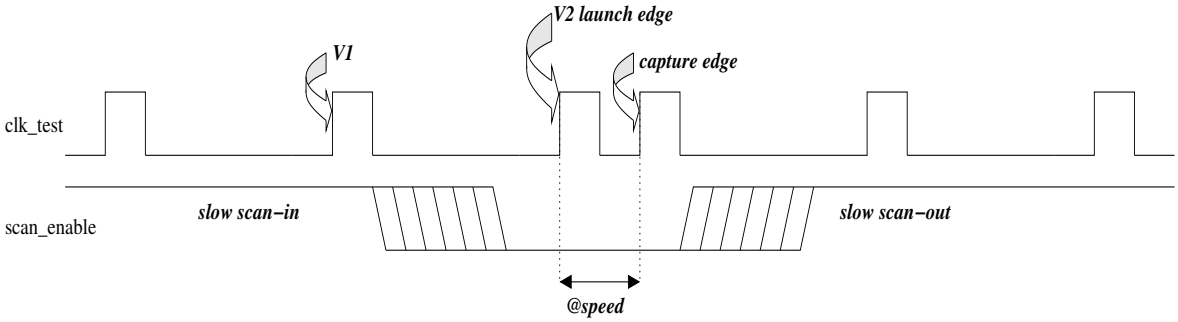


FIG. 9.19: Chronogramme de la technique dite "Launch-On-Capture".

La principale difficulté matérielle avec les tests de type LOLS et LOC réside dans la génération d'une horloge de test. Elle doit être capable de fournir un lancement du vecteur V2 et la capture de la réponse à la fréquence nominale du circuit. L'utilisation d'un testeur externe bas coût n'est pas envisageable. Un mécanisme utilisant la PLL embarquée sur la puce est en général utilisé [Press06, Tendolkar00]. Dû au fait que le micro-testeur est embarqué et que le choix de l'horloge fonctionnelle comme horloge de test a été effectué, la génération de ces deux coups d'horloge à la fréquence nominale ne pose donc pas, à priori, de difficulté.

9.3.3 Implantation dans le micro-testeur

Comme la norme IEEE 1500 ne fournit pas de mécanisme explicite pour le test *at-speed* des coeurs, nous allons présenter ici notre implantation de ces deux techniques dans le micro-testeur.

Implantation du LOLS

Lorsque le FSM du TPIU (celui du Stream Manager plus exactement) décode une instruction HTC d'application d'un vecteur et qu'il a été configuré pour exécuter un test *at-speed* de type LOLS, il parcourt la succession d'états présenté en figure 9.20.

L'automate est dans l'état *IDLE*, le décodage de l'instruction HTC le fait alors passer dans la succession d'état *WAIT*, *SHIFT* et *CAP*, pour finalement revenir dans l'état *IDLE*. Comme nous l'avons vu précédemment dans la section 9.2, l'état *SHIFT* permet d'effectuer un cycle de décalage, l'état *WAIT* permet de temporiser entre deux cycles de décalage. Ainsi, le basculement entre ces deux états permet de gérer une fréquence de scan moins élevée que la fréquence du système. Lors de l'envoi du dernier bit (ou dernier paquet de bits si l'interface parallèle est utilisée), l'automate qui se trouve alors dans l'état *SHIFT*, passe dans l'état *CAP* afin de capturer les réponses en un cycle d'horloge fonctionnelle après le dernier décalage.

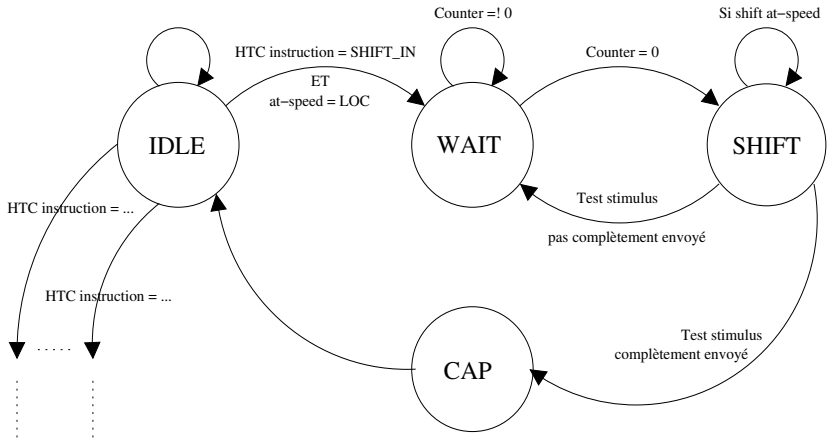


FIG. 9.20: Branche de l'automate du TPIU permettant un test de type LOLS.

La figure 9.21 montre le chronogramme associé à l'exécution de cette séquence pour l'application du vecteur V1, le lancement du vecteur V2 et la capture à la fréquence fonctionnelle. La fréquence de scan est $f_{scan} = f_{sys}/4$. Le chronogramme présente les signaux de gestion d'un coeur ayant de la logique sur l'horloge. Ainsi, les fronts montant sur le signal `clk_int` nous permettent de visualiser l'application du test *at-speed*. Néanmoins, ce type de test *at-speed* peut être également appliqué lorsque le coeur est muni de bascules scan "TEST_MODE".

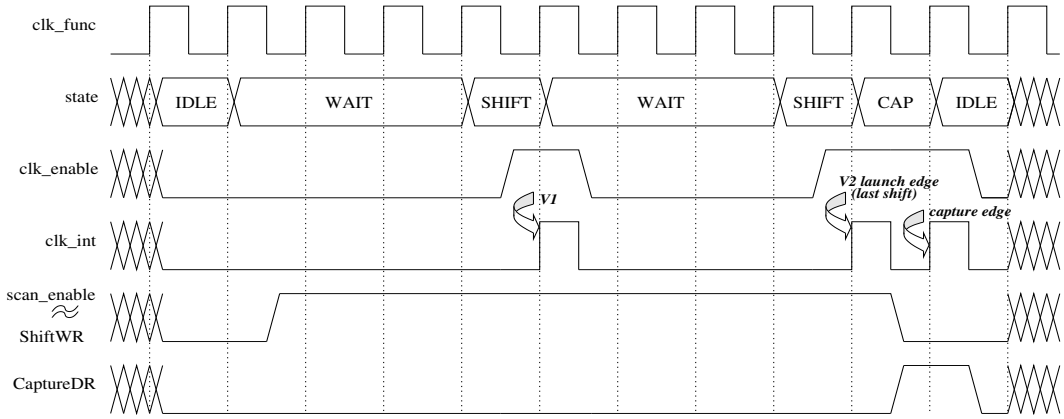


FIG. 9.21: Chronogramme d'un test de type LOLS.

Implantation du LOC

La figure 9.22 présente la branche de l'automate du TPIU décrivant l'application d'un vecteur en utilisant la technique du LOC, du chargement à la capture *at-speed* de la réponse. L'automate est dans l'état *IDLE*, lorsqu'il rencontre l'instruction HTC d'application d'un vecteur et qu'il a été configuré pour exécuter un test *at-speed* de type LOC, il passe dans la succession d'état *WAIT*, *SHIFT*, *WAITLS*, *CAP1* et *CAP2*, pour finalement revenir dans l'état *IDLE*. Comme nous l'avons vu précédemment le basculement entre les deux états *SHIFT* et *WAIT* permet de gérer la fréquence de scan. Après le passage dans l'état *SHIFT* pour l'envoi du dernier bit (ou dernier paquet de bits si l'interface parallèle est utilisée), le vecteur V1 est positionné, l'automate passe dans l'état *WAITLS* afin de laisser le temps au signal `scan_enable` de se stabiliser à la valeur 0 sur toutes les bascules du coeur. Finalement, le passage dans les états *CAP1* et *CAP2* permet de lancer la génération du vecteur V2 (*CAP1*) et de capturer la réponse (*CAP2*) et ceci à la fréquence du circuit.

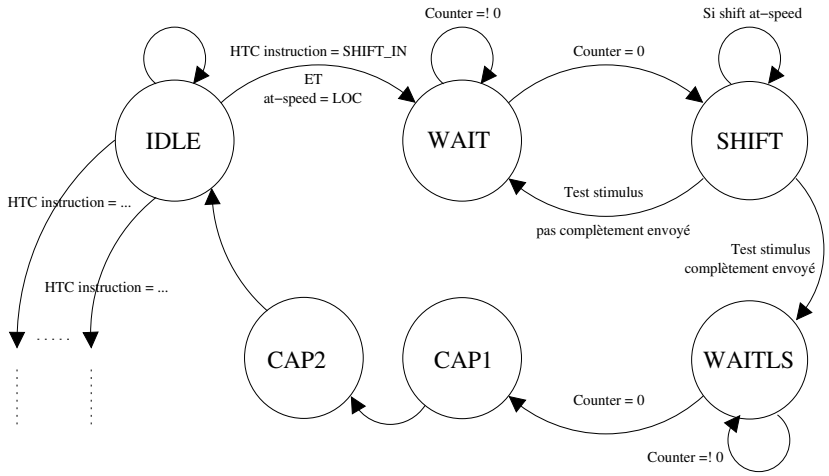


FIG. 9.22: Branche de l'automate du TPIU permettant un test de type LOC.

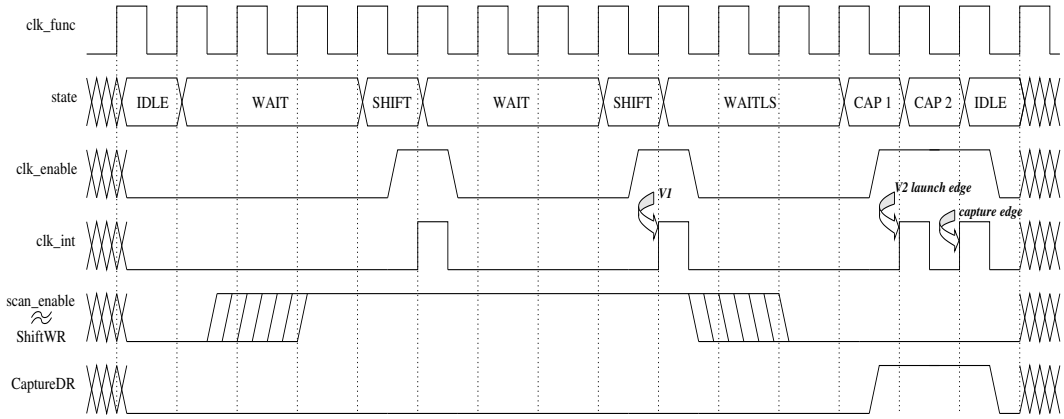


FIG. 9.23: Chronogramme d'un test de type LOC.

La figure 9.23 montre le chronogramme associé à l'exécution de cette séquence pour l'avant dernier décalage de V1, l'application du vecteur V1, le lancement du vecteur V2 et la capture à la fréquence fonctionnelle. Dans ce cas comme dans le précédent nous considérons un circuit avec de la logique sur l'horloge afin de bien repérer les fronts sur l'horloge interne (`clk_int`). Dans cet exemple, la fréquence de scan est, comme pour les autres $f_{scan} = f_{sys}/4$. On remarque que les signaux `scan_enable` et `ShiftWR` peuvent être des signaux "lents".

Ainsi, le micro-testeur peut fournir un test *at-speed* spécifique à chaque coeur. Par exemple, le coeur 1 peut subir un test de type LOLS, alors que le coeur 8 utilise un test de type LOC. Evidemment le coeur doit être fourni avec ses vecteurs de test "*at-speed*" et les supporte (i.e. pour le LOLS le coeur doit avoir un signal scan-enable supportant la transition en une période d'horloge).

Etant donné que seules les techniques de LOC et LOLS sont gérées par les ATPGs, et standards *de facto*, nous nous sommes focalisés uniquement sur l'implantation de ces deux techniques dans le micro-testeur. Néanmoins, nous avons vu que dans [Huang01], le wrapper utilise un mécanisme de test *at-speed* appelé *multiple capture test scheme* (MTS) [Tsai00]. Un pattern est chargé dans la chaîne de scan suivi par plusieurs (k) cycles fonctionnels ou de capture. Ce type de test pourrait tout à fait être conduit par le micro-testeur. Il suffirait pour ce faire, d'avoir un état *CAPN*, à la manière des états *CAP1* et *CAP2* du LOC, mais contrôlé par un compteur pour effectuer les N cycles de capture.

9.4 Conclusion

Ce chapitre nous a permis de déterminer les contraintes de conception des wrappers IEEE 1500 et des coeurs, lorsque le micro-testeur est utilisé.

Nous avons vu, en ce qui concerne les wrappers IEEE 1500 connectés à l'interconnect système, que ces wrappers doivent s'isoler du monde extérieur pendant leur test interne. Ils doivent présenter vers

l'extérieur des valeurs "Safe" afin de ne pas perturber le fonctionnement de l'interconnect système. La présence de ces valeurs "Safe" est recommandée par la norme, elle est obligatoire dans ce cas.

Dans un deuxième temps nous avons vu que le micro-testeur envoie et récupère les vecteurs de test de façon discontinue. Nous avons alors proposé deux stratégies de gestion de ce type de flux. La première étant de modifier les chaînes de scan des coeurs. Nous avons donc proposé l'utilisation d'une bascule scan adaptée à nos besoins : la bascule scan "TEST_MODE". Cette bascule possède, à la différence des bascules conventionnelles, trois modes : le mode fonctionnel, le mode scan et un nouveau mode "Hold" permettant de garder les valeurs dans les bascules. Ainsi deux signaux de commande sont nécessaires à ces bascules. Une autre stratégie pour supporter le flux discontinu est de contrôler l'horloge en faisant de la logique sur l'horloge fonctionnelle. Nous avons pu constater quelles étaient les contraintes temporelles des signaux de test (WSC et WPC) envoyés par le micro-testeur. Dans le cas des nouvelles bascules scan, **tous les signaux de test ont un cycle d'horloge fonctionnelle pour se stabiliser**. Dans le cas de la logique sur l'horloge, **seul le signal "clk_enable" doit se stabiliser en une demie période d'horloge fonctionnelle**. Par contre les autres signaux de test peuvent alors se stabiliser de façon plus lente (plusieurs cycle d'horloge fonctionnelle par exemple).

Finalement ce chapitre nous a permis de traiter le test *at-speed* des coeurs. Nous avons effectué un rappel des types de fautes temporelles (faute de transition et faute de chemin) et des techniques de test associés dans un environnement scan. Les deux techniques présentées, le *Launch-On-Last-Shift* (LOLS) et le *Launch-On-Capture (LOC)* sont gérées par les ATPG. Nous avons vu comment ces deux techniques étaient implantées dans le micro-testeur. Le micro-testeur permet donc un test *at-speed* propre à chaque coeur du système.

Chapitre 10

Résultats expérimentaux

Le micro-testeur exécute les programmes de test HTC de chaque coeur, sous le contrôle du processeur embarqué. Les chapitres précédents ont présenté la stratégie de test et les aspects qualitatifs de l'architecture. Voyons maintenant les aspects quantitatifs.

Voici les questions que l'on est en droit de se poser sur les micro-testeurs en général, et notre architecture en particulier :

- Les micro-testeurs sont-ils capables de rivaliser, en terme de temps de test, avec des approches classiques de type bus dédié, piloté par ATE ?

Le volume des jeux de test est un problème épineux, ainsi on peut se demander pour notre approche :

- Quel est l'impact du format HTC sur les jeux de test ?
- Quel est l'impact de la compaction ?

Comme toutes approches matérielles/logicielles, le micro-testeur ajoute de la surface supplémentaire dédiée au test. Cette surface additionnelle doit se situer dans un intervalle acceptable.

- Quel est la surface supplémentaire engendrée par le micro-testeur ?

Pour répondre à ces questions, des plates-formes de simulation ont été développées afin d'évaluer les performances du micro-testeur. Parmi ces performances on décrira d'abord le **volume des données de test**, calculé grâce à la taille des programmes HTC. Les **temps d'application du test** sont ensuite présentés et comparés à une approche de test classique utilisée chez Philips : TR-Architect. La quatrième section commente la **surface additionnelle** introduite par le micro-testeur. Finalement, la dernière section conclut ce chapitre.

10.1 Les plates-formes de simulation

Les plates-formes de simulation que nous avons réalisées s'appuient sur les benchmarks ITC'02 [Marinissen]. Ces benchmarks sont constitués d'un ensemble de coeurs appelés modules dont les structures de test sont données. Ils offrent la possibilité de faire des comparaisons entre différents TAMs au niveau système.

10.1.1 Les benchmarks ITC'02

Chaque benchmark ITC'02 consiste en un ensemble de M modules. Pour chaque module $m \in M$ nous avons comme informations :

- le nombre d'entrées fonctionnelles i_m,
- le nombre de sorties fonctionnelles o_m,
- le nombre d'entrées/sorties bidirectionnelles fonctionnelles b_m,
- le nombre de chaîne de scan s_m,
- le nombre de bascules $l_{m,k}$ de chaque chaîne de scan k. Ainsi pour chaque module m, le nombre total de bascules scan est $f_m = \sum_{k=1}^{s_m} l_{m,k}$.
- le nombre de patterns de test p_m.

Chaque module peut être vu comme étant un *soft core* ou un *hard core*. En tant que module de type *hard core*, le nombre et la longueur des chaînes de scan ne peuvent être modifiés. Nous dirons que ce module contient des chaînes de scan fixes. Pour les modules de type *soft core*, le nombre de chaînes de scan et les longueurs associées ne sont pas encore déterminés et peuvent donc être optimisés pour l'architecture étudiée. Ainsi, nous dirons que ces modules contiennent des chaînes de scan flexibles.

Le listing 10.1 présente un exemple de fichier benchmark ITC'02. Le benchmark s'appelle p6, et contient 3 modules. Les modules 1 et 2 sont contenus dans le module 0 (top-level). Le module 2 contient 4 chaînes de scan.

Listing 10.1: Exemple de fichier benchmark ITC'02

```
1  SocName p6
2  TotalModules 3
3  Options Power 0 XY 0
4  Module 0 Level 0 Inputs 0 Outputs 0 Bidirs 0 ScanChains 0 :
5  Module 0 TotalTests 0
6  Module 1 Level 1 Inputs 32 Outputs 32 Bidirs 0 ScanChains 0 :
7  Module 1 TotalTests 1
8  Module 1 Test 1 ScanUse 1 TamUse 1 Patterns 12
9  Module 2 Level 1 Inputs 36 Outputs 39 Bidirs 0 ScanChains 4 : 54 53 52 52
10 Module 2 TotalTests 1
11 Module 2 Test 1 ScanUse 1 TamUse 1 Patterns 105
```

10.1.2 Description des plates-formes

Cinq benchmarks ITC'02 ont été utilisés : d695, g1023, p22810, p34392 et p93791. Les benchmarks d695 et g1023 sont fournis par des universités. Le premier contient une dizaine de modules, le deuxième une quinzaine. Les trois benchmarks suivants sont quant à eux fournis par Philips. Ils contiennent entre une vingtaine et une trentaine de modules. La figure 10.1 décrit la plate-forme de simulation correspondant au benchmark d695.

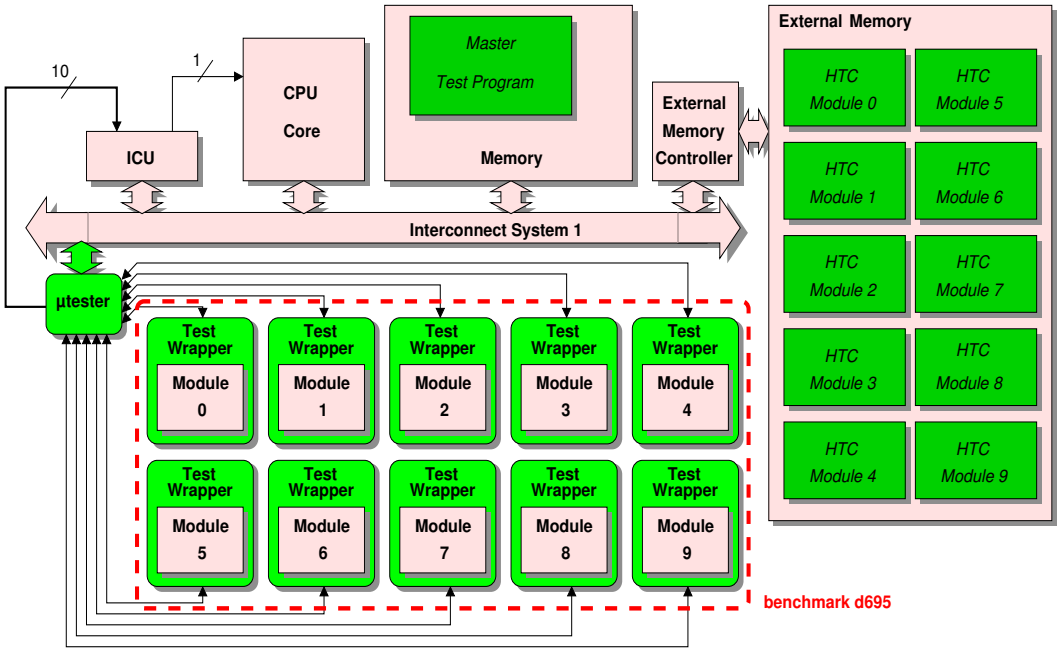


FIG. 10.1: Plate-forme de simulation : le benchmark ITC'02 **d695** contient 10 modules

La plate-forme SystemC

Les plates-formes de simulation sont écrites en SystemC et contiennent à la fois les composants matériels et logiciels.

Les composants matériels. Tous les composants sont écrits en SystemC, précis au bit près et au cycle près (cycle-accurate, bit-accurate CABA) :

- Le processeur ("CPU core" sur la figure 10.1) utilisé est un MIPS R3000.
- Le micro-testeur est constitué de sous-modules respectant la hiérarchie présentée au chapitre 9 (*Prefetch-Buffer*, et *TPIUs* composés chacun d'un *Stream Manager* et d'un *Configuration Manager*).
- Un composant de type mémoire est ajouté pour effectuer le rôle de la RAM embarquée contenant le binaire MIPS.

- Une autre mémoire est ajoutée pour jouer le rôle de RAM externe contenant tous les programmes de test HTC.
- Une ICU (contrôleur d'interruptions) permet de centraliser les IRQs provenant du micro-testeur.
- Tous ces composants sont connectés via un interconnect de type crossbar respectant le protocole VCI [VSIA]. Un interconnect (crossbar, bus, NoC) a pour caractéristiques le débit, la latence, la surcharge etc. Etant donné que notre stratégie utilise ce composant, les caractéristiques de celui-ci influence les temps de test. C'est pourquoi le choix s'est porté sur le crossbar car il permet la connexion point à point des différents composants. Ainsi, il minimise l'introduction de temps parasites dans les résultats sur les temps de test.
- Le mécanisme d'accès au test entre le micro-testeur et les coeurs wrappés est établi grâce à des fils dédiés au protocole IEEE 1500.
- Chaque module du benchmark n'est décrit que par son interface IEEE1500. Chaque module génère un fichier de traces permettant de vérifier que tous les patterns de test ont bien été appliqués.
- Une plate-forme instancie tous ces composants et effectue les connexions entre eux.

Les composants logiciels. Le programme maître (MTP) est écrit en C et compilé par GCC version MIPS. Ce binaire est chargé au démarrage de la simulation SystemC par le composant jouant le rôle de la RAM interne. Son utilisation est minimale. Le programme démarre le test de chaque coeur et attend les interruptions. Pour chaque interruption interceptée, il va chercher les résultats du test (test OK, test KO) dans le TPIU correspondant et les range à une adresse précise en mémoire externe.

Le programme de test de chaque module est écrit en C++ avec la bibliothèque STELA. Les binaires HTC sont générés par l'outil GenSTELA, et chargés au démarrage par le composant RAM externe.

Génération des modules

Comme nous l'avons précisé, chaque module peut être vu comme un *soft core* ou un *hard core*. Les *hard cores* ont des chaînes de scan fixes, en nombre et en longueur. Les *soft cores* ont des chaînes de scan flexibles, leur nombre et leur longueurs peuvent être optimisés pour l'architecture étudiée. Ainsi, pour chaque benchmark utilisé nous disposons de deux plates-formes distinctes :

- La plate-forme dont les modules ont des chaînes de scan fixes.
- La plate-forme dont les modules ont des chaînes de scan flexibles.

Pour les modules à chaînes de scan fixes, la largeur du TAM parallèle (i.e. la largeur WPI/WPO) est égale au nombre de chaînes de scan. Le registre WBR du wrapper IEEE 1500 est réparti (en entrée et en sortie) sur les chaînes de scan afin de former des chaînes de longueurs équivalentes.

Pour les modules à chaînes de scan flexibles, l'optimisation de la largeur des TAMs consiste à avoir un accès parallèle aussi large que possible. Considérant l'architecture interne du micro-testeur, la largeur du TAM doit être choisie dans l'ensemble de valeurs : {32, 16, 8, 4, 2, 1} afin d'éviter autant que possible les bits de bourrage dans les programmes de test.

Les paramètres d'une plate-forme

Chaque plate-forme est générée par un outil en fonction de différents paramètres :

Paramètres des modules. Comme nous l'avons précisé, nous utilisons différents benchmarks, et pour chaque benchmark, les largeurs de TAMs des modules sont calculées suivant trois modes : le mode série (la largeur du TAM est 1 et WSI/WSO sont utilisés), le mode parallèle fixe (la largeur du TAM est égale au nombre de chaînes de scan dans le coeur) et le mode parallèle flexible (la largeur du TAM est optimisée pour l'architecture du micro-testeur).

Paramètres du micro-testeur. Nous pouvons faire varier la taille des FIFOs du Prefetch-Buffer. La fréquence de scan peut elle aussi varier. Nous faisons varier le rapport entre la fréquence de scan et la fréquence du système. Enfin, pour chaque plate-forme, le mode compaction du micro-testeur peut être activé ou désactivé.

Pour synthétiser, voici les différents paramètres et leur valeurs :

- le benchmark : d695, g1023, p22810, p34392, p93791.
- la fréquence $f_{scan} = f_{sys}/x$ avec $x \in [1, 2, 4, 8, 10, 16, 20, 30, 32]$.
- la taille des FIFOs du PB : 64, 32, 16, 8, 4.
- le mode du TAM : série, parallèle fixe, parallèle flexible.
- la compaction : activée ou désactivée

Les résultats qui sont présentés par la suite représentent une synthèse de ces 1800 simulations.

10.2 Volume des données de test

Un programme de test HTC contient les stimuli de test, les réponses de test ainsi que les informations de contrôle.

Puisque la taille d'un stimulus de test correspond au nombre de bascules scan internes (f_m) plus le nombre d'entrées fonctionnelles (i_m et b_m) d'un module m, la taille totale, en bits, d'un stimulus est $ts_m = (i_m + b_m + f_m)$. L'ensemble des stimuli, correspond à la taille d'un stimulus multiplié par le nombre de pattern p_m. Soit $TS_m = ts_m * p_m$. De la même façon, la taille d'un vecteur de test réponse correspond au nombre de cellules scan internes plus le nombre de sorties fonctionnelles (o_m et b_m). Ainsi, la taille totale occupée par les réponses est $TR_m = (o_m + b_m + f_m) * p_m$. Pour chaque module m le volume total de données de test est $v_m = TS_m + TR_m$. Pour un benchmark b, le volume de test total est $V_b = \sum_{m=1}^{M} v_m$. Ce volume représente les données de test "brutes", puisqu'aucune autre information comme le contrôle n'est introduite. Nous nous comparerons par la suite à ces données de test brutes.

En ce qui concerne les programmes de test HTC, ils contiennent les stimuli de test comme décrit ci-dessus, encapsulés dans les instructions de test HTC (voir section 8.1.2). Les réponses de test TR_m

du module m peuvent soit être ajoutées dans le programme HTC (et sont donc encapsulées dans des instructions HTC) soit être réduites à une signature de 32 bits si la compaction est activée. Nous allons voir tout d'abord l'influence de l'encapsulation des vecteurs de test au format HTC, puis nous verrons dans quelles proportions la compaction permet de réduire la taille des programmes.

10.2.1 Influence du format HTC sur le volume des données

Le tableau 10.1 présente l'augmentation due au format HTC par rapport au volume des données de test brutes, pour des modules à chaînes de scan fixes dans un cas, à chaînes de scan flexibles dans l'autre.

Chaînes de scan	d695	g1023	p22810	p34392	p93791	Moyenne
Fixes	7.48%	27.32%	61.61%	118.28%	42.94%	**51.5%**
Flexibles	6.94%	24.00%	19.75%	20.99%	4.05%	**15%**

TAB. 10.1: Augmentation du volume des données de test brutes dû au format HTC.

L'augmentation dû au format HTC sur le volume des données brut est attribuable à deux facteurs principaux :

- Les instructions HTC
- Les bits de bourrage

Influence des instructions HTC

L'influence stricte de l'ajout des instructions HTC peut être observée dans le cas de chaînes de scan flexibles. En effet, dans ce cas la largeur du TAM est optimisée pour minimiser le nombre de bits de bourrage. Les valeurs se situent entre 4% et 24%. Les variations sont dues à la taille des vecteurs ainsi qu'à leur nombre. En effet, une instruction HTC est nécessaire à l'application d'un vecteur. Ainsi, si les modules ont beaucoup de vecteurs et/ou s'ils sont courts, beaucoup d'instructions HTC seront ajoutées. Si les modules ont peu de vecteurs et/ou s'ils sont longs, peu d'instructions HTC seront ajoutées. En moyenne les instructions HTC ajoutent **15%** au volume des données brutes.

Influence des bits de bourrage

L'influence des bits de bourrage peut être observée dans le cas des chaînes de scan fixes. En effet, si le nombre de chaînes de scan n'est pas un diviseur de 32, des bits de bourrage sont ajoutés dans le programme. D'ailleurs on peut remarquer le peu de différence entre chaînes de scan fixes et flexibles pour le benchmark d695 (moins de 1%). Ce phénomène s'explique par le fait que ce benchmark ne contient que des modules dont le nombre de chaînes de scan est divisible par 32. De même pour le g1023, qui ne contient qu'un seul module dont le nombre de chaînes de scan n'est pas divisible par 32, au contraire des benchmarks Philips (p22810, p34392 et p93791). Le cas du benchmark p34392 est

intéressant. On peut voir que les instruction HTC ajoutent environ 20% aux données (cas des chaines de scan flexibles dans le tableau 10.1). On peut donc conclure que dans ce cas, les bits de bourrage ajoutent environ 100% aux données (118%-20%).

Ces différences entre les chaînes de scan fixes et flexibles montrent bien la sensibilité de notre architecture à la largeur du TAM.

10.2.2 Influence de la compaction sur le volume des données

Le tableau 10.2 présente la réduction du volume des données (et non l'augmentation) due à la capacité de compaction du micro-testeur, par rapport au volume des données de test brutes. Les réponses de test sont réduites seulement à une signature sur 32 bits.

Chaînes de scan	d695	g1023	p22810	p34392	p93791	Moyenne
Fixes	46.26%	26.10%	9.22%	-23.79%	26.08%	**16.8%**
Flexibles	46.91%	32.21%	35.88%	32.02%	45.87%	**38.6%**

TAB. 10.2: Réduction du volume des données due à la compaction, par rapport au volume des données de test brutes.

On peut remarquer dans ce tableau une augmentation de 23.79% pour le benchmark p34392 dans le cas de chaînes de scan fixes par rapport au volume des données de test brutes. Ceci est dû au fait que sans compaction, le programme HTC augmentait déjà de plus de 2 fois le volume du jeu de test (voir tableau précédent 10.1). Ainsi, même s'il y a réduction, elle s'opère sur un programme déjà hypertrophié. On peut noter tout de même, que la compaction permet de réduire, en moyenne, de 45% le même programme de test sans compaction.

La réduction du volume, en moyenne, pour les architectures à chaînes de scan fixes est d'environ **16.8%**. Pour les architectures à chaînes de scan flexibles, la réduction est d'environ **38.6%**.

La taille du programme de test maître (MTP), conçu en implémentant un algorithme minimal, consistant juste à lancer les tests, et à récupérer en fin de test le statut de chaque coeur (présenté en annexe D.4) est négligeable ($\approx$ 1 Ko), comparée à la taille totale des programmes de test HTC (de l'ordre de plusieurs dizaine de millions de bits, présentée en annexe dans le tableau D.8).

10.2.3 Conclusion

Cette section a pointé le fait que l'architecture fixe de 32 bits est très sensible à la largeur du TAM des coeurs. Alors que le format HTC peut dans certains cas n'ajouter qu'à peine **4%** au jeu de test brut (**15%** en moyenne), les bits de bourrage peuvent, quant à eux, quasiment **doubler** le jeu de test originel. L'influence du format HTC est minimisée lorsque la taille du TAM est divisible par 32 (pas de bits de bourrage) et lorsque les vecteurs de test sont longs (minimise l'incidence de l'insertion d'instructions HTC dans le programme de test).

Néanmoins, pour répondre à la question de l'impact du micro-testeur sur le volume du jeu de test brut, nous pouvons répondre que les capacités de compaction du micro-testeur permettent de réduire

en moyenne de **27.7%** le volume des données (chaînes de scan fixes et flexibles confondues) et peut atteindre jusqu'à **45%** de réduction.

10.3 Temps d'application du test

Nous avons vu que le micro-testeur possède la capacité de compacter les réponses de test. Lorsque la compaction est désactivée, les réponses peuvent être comparées bit par bit à celles attendues. La désactivation de la compaction n'a donc d'intérêt que dans une optique de diagnostic. En test de production, l'objectif est de tester le plus rapidement possible. C'est pourquoi dans cette section, tous les temps présentés sont obtenus lorsque la compaction est activée.

Dans cette section, nous allons tout d'abord étudier les influences de trois différents facteurs sur le temps de test : la fréquence de scan, la largeur du TAM et la taille des FIFOs du Prefetch-Buffer.

Ensuite, nous comparerons les temps de test du micro-testeur à ceux d'un TAM traditionnel, piloté par un testeur externe, développé par Philips : TR-Architect [Goel02].

10.3.1 Influence de la fréquence de scan

Le temps d'application du test dépend de la fréquence d'application du test.

Dans les approches SBST *fonctionnelles* tel que [Corno01b, Chen01b] la fréquence d'application du test est celle du système : f_{sys}, fréquence d'horloge fonctionnelle.

Dans les approches traditionnelles où le TAM est piloté par le testeur externe tels que le TestBus [Varma98] ou le TestRail [Marinissen98] la fréquence opératoire est la fréquence du testeur externe. Cette fréquence est celle utilisée pour le décalage des bascules scan f_{scan}, qui est généralement plus lente que f_{sys} (cf section 9.2 sur les aspects temporels).

Dans les approches SBST *structurelles* telles que [Wang04] ou celle que nous avons présentées dans ce manuscrit, les deux types de fréquence sont utilisés. Pendant que l'ensemble du système fonctionne à f_{sys}, les bascules scan, dans tous les coeurs testés, sont décalées à la fréquence f_{scan}. Dans [Wang04], les auteurs présentent seulement le temps de test en utilisant deux valeurs de f_{scan} différentes (à la vitesse du système et à la moitié). Dans cette section, nous allons présenter des résultats pour plusieurs valeurs de f_{scan}. Les résultats sont présentés pour les ratios $\frac{f_{sys}}{f_{scan}}$ suivants : 32, 16, 8, 4, 2, et 1.

La figure 10.2 présente l'évolution du temps de test (en nombre de cycles fonctionnels) en fonction de la vitesse de scan. Le benchmark utilisé est le d695, la taille des FIFOs du Prefetch-Buffer est de 32 mots. La courbe à carrés pleins, représente le nombre de cycles fonctionnels nécessaires au micro-testeur pour effectuer le test de l'ensemble des modules, pour le cas où ces derniers sont à chaînes de scan fixes et que nous appellerons cas "fixe" par la suite. La courbe à carrés vides, représente le cas où les modules sont à chaînes de scan flexibles et que nous appellerons cas "flexible" par la suite.

La première conclusion que l'on peut tirer, est que, comme on l'attendait, plus la fréquence de scan est lente, plus le temps de test augmente.

La deuxième conclusion, elle aussi attendue, est que plus la fréquence de scan diminue, plus l'écart en temps de test entre le cas fixe et le cas flexible s'amplifie.

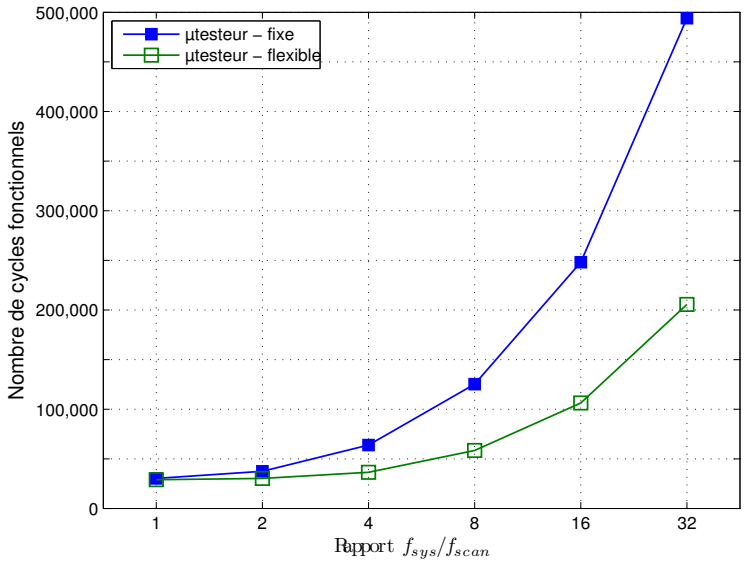


FIG. 10.2: Evolution du temps de test du micro-testeur en fonction de la vitesse de scan. Le benchmark utilisé est le d695.

10.3.2 Influence de la taille des FIFOs du Prefetch-Buffer

Le Prefetch-Buffer du micro-testeur nourrit des FIFOs qui sont utilisées comme "mémoire cache" contenant les programmes HTC (cf. section 9.1.2). Le Prefetch-Buffer, pour remplir ces FIFOs, fait des requêtes de lecture en rafale sur l'interconnect système. La taille de ces rafales dépend de la taille de la FIFO. Plus la FIFO est profonde, plus la rafale sera grande, permettant ainsi d'augmenter le débit.

Les FIFOs peuvent avoir comme profondeur : 64, 32, 16, 8 et 4 mots. Pour chaque valeur de la fréquence de scan f_{scan}, les 5 tailles des FIFOs ont été simulées. Le tableau 10.3 présente le nombre de cycles nécessaires au micro-testeur pour achever le test du benchmark d695 en fonction des différentes tailles des FIFOs. Les paramètres de cette simulation sont : les modules sont à chaînes de scan flexibles et une fréquence de scan 4 fois moins élevée que la fréquence du système. On peut constater, de manière générale, que plus les FIFOs sont profondes plus le test est rapide. On peut voir que pour des FIFOs de 32 mots, le test est deux fois plus rapide que pour des FIFOs de 4 mots.

Sur l'ensemble des simulations, on peut constater que l'impact des FIFOs sur le temps de test diminue avec le ralentissement de la fréquence de scan. La figure 10.3 présente cette évolution sur l'ensemble des benchmarks pour les cas à chaînes de scan fixes (carrés) et à chaînes de scan flexibles (ronds). Un point du graphique correspond à l'écart moyen entre les temps de test obtenus par des FIFOs de tailles différentes, pour une fréquence de scan donnée.

Tailles des FIFOs	**64**	**32**	**16**	**8**	**4**
Nombre de cycles	35,263	36,449	40,283	50,722	74,483

TAB. 10.3: Nombre de cycles en fonction de la taille des FIFOs du Prefetch-Buffer. Le benchmark est le d695 à chaînes de scan flexibles, et $f_{scan} = f_{sys}/4$.

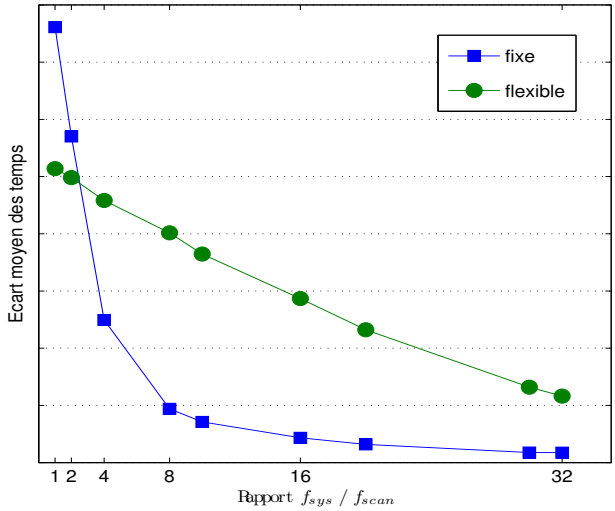


FIG. 10.3: Impact de la taille des FIFOs en fonction de la fréquence de scan

Les deux courbes sont décroissantes, indiquant que plus la fréquence de scan est basse, moins les temps de test pour les différentes tailles des FIFOs sont éloignés. Ce qui signifie que l'impact de la taille de FIFOs diminue avec la fréquence de scan. L'évolution pour les chaînes de scan flexibles reste linéaire. On peut remarquer sur ce graphique l'évolution pour les modules à chaînes de scan fixes. Deux parties peuvent être distinguées. On peut voir que lorsque la fréquence de scan devient 8 fois moins élevée que la fréquence système, la taille des FIFOs ne joue plus du tout. A l'inverse, lorsque le rapport entre f_{sys} et f_{scan} est inférieur à 8, la taille des FIFOs impacte les temps de test. Par exemple, si un système fonctionne à 200MHz et que la fréquence de scan de l'ensemble des modules testés est de 20MHz, la taille des FIFOs n'a pas d'impact sur le temps de test. On préférera alors des FIFOs petites qui ont le mérite d'occuper moins de place sur silicium. Par contre, si la fréquence de scan est de 100MHz, des FIFOs profondes permettront d'accélérer le test.

10.3.3 Comparaison avec TR-Architect

Nous avons vu quels étaient les différents facteurs influençant les temps de test du micro-testeur (taille des FIFOs, fréquence de scan etc.). Nous allons dans cette partie comparer les temps de test du micro-testeur avec les temps de test d'un TAM traditionnel piloté par un testeur externe.

TR-Architect est le logiciel permettant l'optimisation du TAM TestRail utilisé chez Philips [Goel02].

Ce type de bus est une approche classique guidée par un testeur externe. L'article [Goel02] présente les temps de test, en nombre de cycles, obtenu par TR-Architect pour différents benchmarks ITC'02. La largeur de ce TAM pouvant varier, nous avons sélectionné pour nous comparer, les résultats pour un TAM de 32 bits de large. Dans cette configuration, TR-Architect utilise 64 plots d'entrées/sorties (32 scin et 32 scout) dédiés au TAM, ce qui correspond à la taille de notre interface avec la RAM externe (32 bits de données + 32 bits d'adresses).

Le tableau 10.4 présente les résultats obtenus par le micro-testeur (colonne 2) et TR-Architect (colonne 3). Les paramètres sont f_{scan} = f_{sys}, les profondeurs des FIFOs dans le micro-testeur sont de 32 mots et nous sommes dans le cas où les chaînes de scan sont fixes. Nous pouvons constater que le rapport entre le micro-testeur et TR-Architect varie entre 1.5 et 3.5 et est en faveur de TR-Architect. La moyenne est de **2.47**. Dans cette configuration TR-Architect est plus rapide que le micro-testeur.

Benchmark	**μtesteur**	**TR-Architect**	$\frac{\mu testeur}{TR-Architect}$
d695	30,270	21,690	1.4
g1023	51,233	16,855	3
p22810	799,166	226,640	3.53
p34392	1,436,175	542,746	2.6
p93791	1,709,747	940,745	1.82

TAB. 10.4: Nombre de cycles nécessaires au micro-testeur et à TR-Architect pour effectuer le test de différents benchmarks. Les paramètres sont f_{scan} = f_{sys} et les chaînes de scan sont fixes.

Le tableau 10.5 présente les résultats obtenus pour le même contexte mais dans ce cas, les chaînes de scan sont flexibles. Le rapport entre le micro-testeur et TR-Architect est toujours en faveur de ce dernier, mais dans ce cas le micro-testeur se rapproche des performances de TR-Architect. La moyenne est de **1.54**.

Benchmark	**μtesteur**	**TR-Architect**	$\frac{\mu testeur}{TR-Architect}$
d695	28,972	21,503	1.35
g1023	29,919	16,795	1.78
p22810	363,107	223,368	1.63
p34392	795,507	505,783	1.57
p93791	1,259,093	914,456	1.38

TAB. 10.5: Nombre de cycles nécessaires au micro-testeur et à TR-Architect pour effectuer le test de différents benchmarks. Les paramètres sont f_{scan} = f_{sys} et les chaînes de scan flexibles.

Ces deux derniers tableaux présentent les résultats lorsque f_{scan} = f_{sys}. Ceci signifie deux choses. La première est que l'ensemble des modules peuvent subir un scan à la fréquence nominale (at-speed). La deuxième concerne TR-Architect, et suppose qu'un testeur externe performant est disponible, afin de piloter le TAM à la fréquence du système.

Le tableau 10.6 présente les résultats obtenus par le micro-testeur et TR-Architect pour une fréquence

de scan égale à la moitié de la fréquence du système ($f_{sys}/f_{scan} = 2$). Les modules sont considérés à chaînes de scan fixes. Le nombre de cycles affichés représente le nombre de cycles fonctionnels nécessaires à l'achèvement du test. TR-Architect fonctionnant uniquement à la fréquence f_{scan} nous avons compté deux cycles fonctionnels pour un cycle de scan. On peut remarquer sur le benchmark d695 que le micro-testeur est plus rapide que TR-Architect. Il reste que dans l'ensemble le micro-testeur est un peu moins performant que TR-Architect, la moyenne étant de **1.85**.

Benchmark	**μtesteur**	**TR-Architect**	$\frac{\mu testeur}{TR-Architect}$
d695	37,424	43,380	0.86
g1023	86,227	33,710	2.56
p22810	1,362,293	453,280	3
p34392	1,731,969	1,085,492	1.6
p93791	2,330,437	1,881,490	1.24

TAB. 10.6: Nombre de cycles nécessaires au micro-testeur et à TR-Architect pour effectuer le test de différents benchmarks. Les paramètres sont $f_{scan} = f_{sys}/2$ et les chaînes de scan fixes.

Finalement, le tableau 10.7 présentant les résultats dans un contexte où les chaînes de scan sont flexibles, montre que le micro-testeur est plus performant que TR-Architect.

Benchmark	**μtesteur**	**TR-Architect**	$\frac{\mu testeur}{TR-Architect}$
d695	30,216	43,006	0.7
g1023	33,691	33,590	1
p22810	396,759	446,736	0.89
p34392	837,317	1,011,566	0.83
p93791	1,328,801	1,828,912	0.73

TAB. 10.7: Nombre de cycles nécessaires au micro-testeur et à TR-Architect pour effectuer le test de différents benchmarks. Les paramètres sont $f_{scan} = f_{sys}/2$ et les chaînes de scan flexibles.

Les courbes présentées en figures 10.4, 10.5, 10.6, 10.7 et 10.8 montrent l'évolution du temps de test pour le micro-testeur et TR-Architect (pour les cas "fixe" et "flexible") en fonction de la fréquence de scan. Au regard des ces cinq graphiques, nous pouvons tirer les conclusions suivantes :

- TR-Architect est peu sensible au cas "fixe" et au cas "flexible" (les deux courbes sont en général confondues).
- Le micro-testeur est très sensible au cas "fixe" et au cas "flexible".
- TR-Architect (cas "fixe" ou "flexible") est globalement meilleur que le micro-testeur en version "fixe".
- Le micro-testeur en version "flexible", est globalement meilleur que TR-Architect.

Pour conclure, nous pouvons rappeler que le micro-testeur n'est contraint en fréquence de scan que par les capacités du circuit, alors que TR-Architect est en plus, contraint par la fréquence que peut supporter le testeur externe. Prenons un exemple concret, le test du benchmark p34392 où les coeurs

sont à chaînes de scan fixes. Nous supposons un système fonctionnant à la fréquence de 200MHz, et dont la fréquence maximale de scan est de 100MHz (pour tous les coeurs). Le testeur externe peut supporter une fréquence de test de 25MHz maximum.

- Il faut donc au micro-testeur 8.5 milli-seconde pour effectuer le test.
- Il faut à TR-Architect 22 milli-seconde pour effectuer le même test.

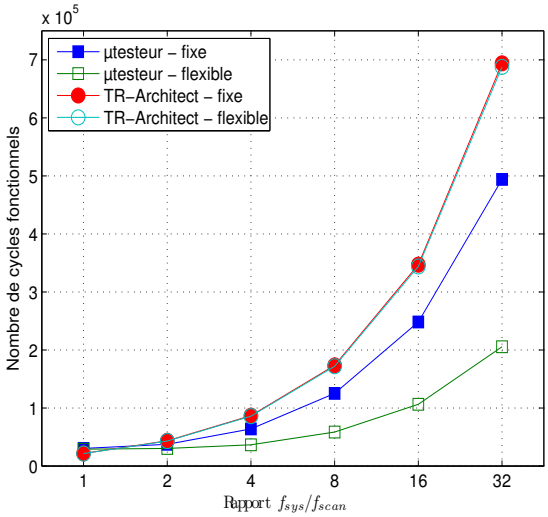


FIG. 10.4: Evolution du temps de test en fonction de la vitesse de scan pour le benchmark d695.

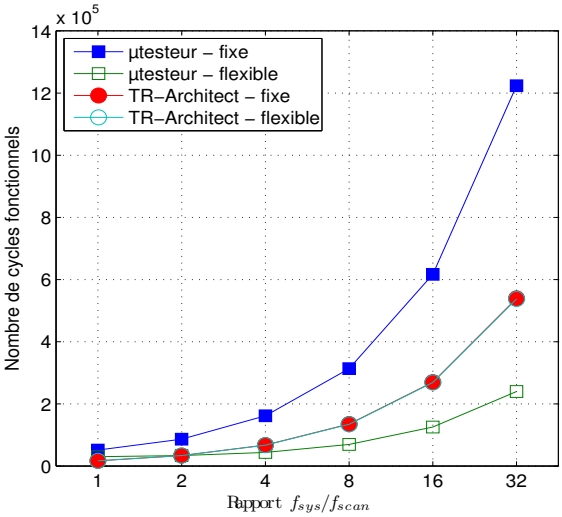


FIG. 10.5: Evolution du temps de test en fonction de la vitesse de scan pour le benchmark g1023.

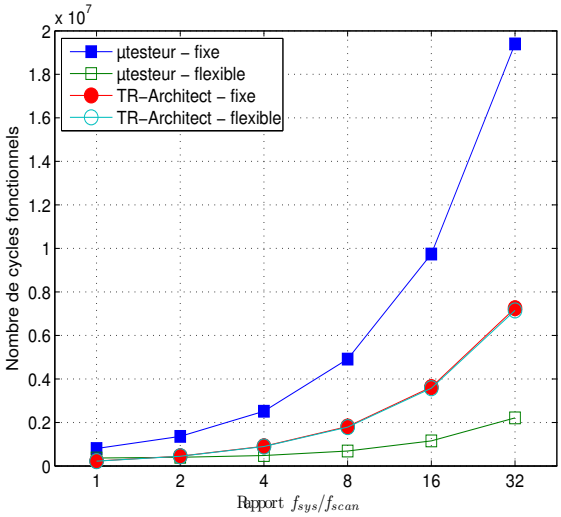


FIG. 10.6: Evolution du temps de test en fonction de la vitesse de scan pour le benchmark p22810.

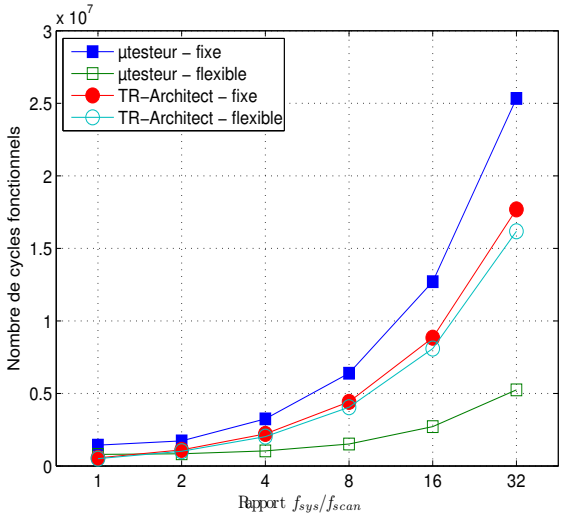


FIG. 10.7: Evolution du temps de test en fonction de la vitesse de scan pour le benchmark p34392.

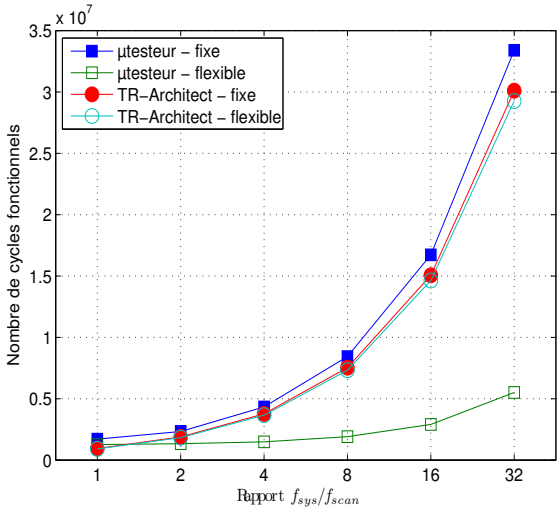


FIG. 10.8: Evolution du temps de test en fonction de la vitesse de scan pour le benchmark p93791.

10.3.4 Conclusion

L'utilisation des benchmarks ITC'02 nous a permis de voir les performances du micro-testeur en terme de temps de test. Nous avons vu quels étaient les différents facteurs influençant les temps de test du micro-testeur :

- La fréquence de scan. En effet, le test est effectué en utilisant deux types de fréquences. La fréquence à laquelle les coeurs sont testés (f_{scan}) et la fréquence à laquelle le système opère (f_{sys}). Nous avons pu constater que plus la fréquence de test diminue plus le test est long.
- La taille des FIFOs dans le Prefetch-Buffer. La taille de ces tampons n'influence les temps de test que lorsque la fréquence de test est élevée (jusqu'à f_{sys}/8).
- La largeur des TAMs. A travers l'utilisation des coeurs à chaînes de scan fixes et à chaînes de scan flexibles, nous avons pu constater que les temps de test du micro-testeur étaient meilleurs pour ce deuxième cas (flexible). Les coeurs à chaînes de scan flexibles permettent une optimisation de largeur du TAM, ce qui signifie dans le cas du micro-testeur une largeur de TAM divisible par 32. Lorsque la largeur du TAM n'est pas divisible par 32, des bits de bourrage sont insérés dans le programme HTC, gonflant la taille de celui-ci et augmentant par conséquence les temps de test.

L'utilisation des benchmarks ITC'02 nous a permis de comparer notre architecture à un TAM traditionnel piloté par un testeur externe : TR-Architect de Philips. Nous avons pu constater que TR-Architect est plus rapide que le micro-testeur lorsque la fréquence de test est égale à la fréquence nominale du circuit. Mais ceci impose que le testeur externe puisse piloter un test à cette cadence et que le circuit puisse également le supporter. Or le micro-testeur n'est pas contraint par les performances du testeur externe. Dans le cas où la fréquence du testeur externe est plus faible que celle du système, le micro-testeur est alors plus performant.

Nous pouvons comparer notre approche au micro-testeur utilisé dans [Lee05]. Dans cette approche, présentée dans l'état de l'art section 7.3.3, le micro-testeur appelé "TAM controller" utilise un bus de test pour envoyer les données aux différents coeurs testés. Les temps de test sont présentés pour une plate-forme contenant dix coeurs. La taille totale des stimuli représente 0.754 millions de bits. Le "TAM controller" nécessite 5 millions de cycles pour exécuter le test en mode compaction. Nous pouvons prendre, afin de nous comparer, le benchmark d695 qui contient dix modules et dont la somme des stimuli représente 0.584 millions de bits. Dans le cas ou les coeurs ont des chaînes de scan fixes, le micro-testeur teste tous les modules en environ 30 000 cycles. Le test du benchmark p93791 représente une taille de stimuli de quasiment 28 millions de bits pour une trentaine de coeurs. Le micro-testeur exécute le test en 1,5 millions de cycles. Notre architecture est beaucoup plus rapide. Ceci est dû, principalement, au test concurrent de chaque coeur et de l'accès DMA à la RAM externe.

10.4 Surface additionnelle

La surface additionnelle est un point crucial en DfT.

Afin de connaître la surface engendrée par le micro-testeur un modèle VHDL a été réalisé. La suite d'outils SYNOPSYS [SYNOPSYS] a été utilisée pour la synthèse de ce modèle. La bibliothèque de cellules standards utilisée est SXLIB [ALL].

10.4.1 Surface globale

Le modèle conçu pour la génération du layout contient un Prefetch-Buffer avec huit TPIUs. Ainsi, ce micro-testeur peut tester huit coeurs différents. La taille des FIFOs est de 32 mots. Le layout a été généré avec la suite d'outils Silicon Ensemble [CADENCE] en technologie $0.13\mu m$. **La surface complète de ce micro-testeur est de 0.53mm^2.**

10.4.2 Surface supplémentaire par IP

Puisque le micro-testeur permet le test de huit coeurs différents, **la surface additionnelle pour le test d'un coeur est de 0.066mm^2**.

Afin d'obtenir le pourcentage de surface supplémentaire nous devons prendre en compte l'aire du coeur testé. La figure 10.9 présente le pourcentage de surface additionnelle du micro-testeur en fonction de la surface du coeur considérée. Dans les techniques de DfT classiques une surface additionnelle jusqu'à 10% est largement acceptée. Ainsi, le micro-testeur peut être utilisé pour tester des coeurs ayant une surface minimale de $0.65mm^2$. Par exemple, selon la courbe représentée dans la figure 10.9 si le micro-testeur est utilisé pour tester des coeurs dont la surface est de $1.5mm^2$ **la surface supplémentaire ajoutée par le micro-testeur est de 4.4%**. Pour donner un ordre de grandeur, un processeur embarqué MIPS d'entrée de gamme, le MIPS32 4KEc avec des mémoire caches de 8Ko/8Ko, synthétisé en $0.13\mu m$ TSMC à une taille de $2.5mm^2$ [MIPS].

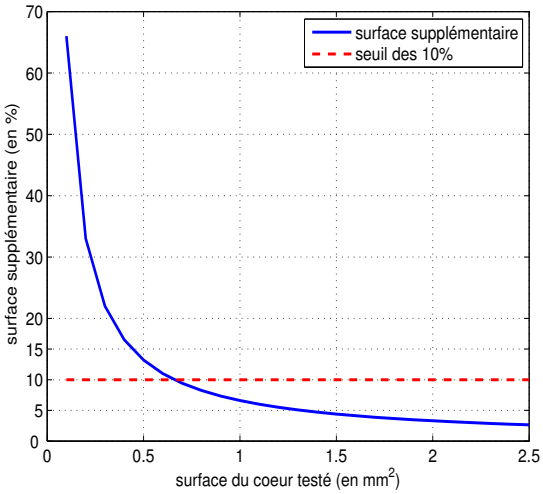


FIG. 10.9: Surface supplémentaire (en %) introduite par le micro-testeur en fonction de la surface du coeur testé.

10.5 Conclusions

Ce chapitre nous a permis d'explorer les performances du micro-testeur. Nous avons abordé les trois volets que sont, l'impact du format du jeu d'instructions du micro-testeur (HTC) sur le volume des jeux de test, les temps de test grâce aux benchmarks ITC'02 et la surface engendrée par le micro-testeur.

Le volume des jeux de test

Nous avons pu constater dans cette section l'impact du format HTC, l'impact des bits de bourrage introduit dans les programmes de test, ainsi que l'impact de la compaction.

L'influence stricte des instructions HTC, sur le volume des données de test brutes représente moins de 15% du programme. Lorsque la largeur du TAM n'est pas optimisée pour l'architecture du micro-testeur, des bits de bourrage sont insérés dans le programme. Ainsi, les résultats sur les cinq benchmarks ITC'02 utilisant des coeurs de type *hard cores* (chaînes de scan fixes) montrent que le passage au format HTC (instructions et bits de bourrage) des données de test brutes représente une augmentation d'environ 50%. Lorsque des *soft cores* (chaînes de scan flexibles et donc optimisées) sont utilisés, l'augmentation du volume des données de test se situe autour de 15%. Pour synthétiser, si l'on considère que les coeurs utilisés sont pour moitié des *hard cores* et pour l'autre moitié des *soft cores*, le formatage HTC augmente le volume des données de test brutes de 33%.

D'un autre côté, si l'on prend en compte la capacité du micro-testeur à compacter les réponses, le volume des données de test brutes est réduit de 27.7% en moyenne (*hard cores* et *soft cores* confondus). Cette réduction peut atteindre jusqu'à 47% dans certains cas.

Les temps d'application du test

Le temps d'application du test dépend de la fréquence d'application du test. Notre approche utilise deux types de fréquences. Le système fonctionne à la fréquence nominale du circuit, pendant que le micro-testeur teste les coeurs à une fréquence de test inférieure ou égale à la fréquence nominale.

Nous avons pu voir dans cette section quels étaient les différents facteurs influençant le test : fréquence de scan, profondeur des FIFOs dans le Prefetch-Buffer et largeur du TAM. Cette section a aussi été l'occasion de comparer notre approche à un TAM traditionnel piloté par un testeur externe : TR-Architect. Cet outil, développé par Philips, permet l'optimisation du TAM TestRail en fonction des caractéristiques du SoC. A fréquence de test égale, dans le cas où tous les coeurs sont de type *hard cores*, TR-Architect est en général plus performant que le micro-testeur. Lorsque les coeurs testés sont de type *soft cores*, le micro-testeur obtient généralement de meilleures performances. Néanmoins, TR-Architect est contraint par les capacités du testeur externe, ce qui n'est pas le cas du micro-testeur.

Nous avons aussi comparé notre approche au micro-testeur décrit dans [Lee05]. Nous avons pu constater que notre approche était plus performante en terme de temps de test.

Surface engendrée par le micro-testeur

La conception modulaire du micro-testeur permet de rendre sa surface proportionnelle au nombre de coeurs testés. La taille du micro-testeur pour tester 8 coeurs différents est égale à $0.53mm^2$ en technologie $0.13\mu m$. Un processeur MIPS32 4KEc embarqué dans les SoCs avec ses caches occupe une taille de $2.5mm^2$ dans la même technologie. La surface engendrée par le micro-testeur nécessaire au test de ce coeur représente une augmentation de surface de 2.6%.

Pour synthétiser, nous pouvons dire que le micro-testeur permet de réduire le volume des données de test d'environ 27% grâce aux capacités de compaction. Il permet d'effectuer le test d'un ensemble de coeurs rapidement, puisque le micro-testeur n'est pas contraint par les limites du testeur externe. Finalement la surface engendrée par ce composant est faible.

Chapitre 11

Conclusion

L'approche SBST permet, de par son côté embarqué, d'effectuer un test de qualité et, de par son côté logiciel, permet de minimiser la surface supplémentaire dédiée au test en permettant la réutilisation des coeurs programmables dans les SoCs. D'un autre côté, aujourd'hui les coeurs sont souvent livrés compatibles avec la norme IEEE 1500. Nous avons donc proposé une stratégie SBST permettant le test des coeurs compatibles avec le standard IEEE 1500.

Après avoir étudié les différentes stratégies SBST pour le test des composants d'un SoC, nous avons pu constater que l'approche utilisant un composant dédié au test permettait une plus grande flexibilité que les approches par wrappers uniquement ou celles sans introduction de DfT (fonctionnelles). L'approche par micro-testeur nous permet d'être compatible avec la norme IEEE 1500.

L'analyse des avantages et des inconvénients des stratégies utilisant des micro-testeurs, proposées dans la littérature, nous a permis d'élaborer une architecture de micro-testeur complètement modulaire. Cette analyse nous a aussi poussé à utiliser une mémoire externe au SoC stockant les programmes de test des différents coeurs à vérifier.

Le micro-testeur exécute les programmes de test stockés en mémoire externe sous le contrôle du processeur embarqué. Le processeur est en charge du processus de test au niveau système, pendant que le micro-testeur se préoccupe du test au niveau coeur. Nous avons proposé un micro-testeur compatible avec la norme IEEE 1500, néanmoins, l'architecture modulaire de ce composant permet au micro-testeur d'adapter la stratégie de test en fonction du protocole de test utilisé par chaque coeur (JTAG, IEEE 1500, contrôleur BIST etc.). De plus, le test *at-speed* étant aujourd'hui nécessaire afin d'obtenir un test de qualité, nous avons introduit dans le micro-testeur les fonctionnalités permettant d'effectuer un test de type *Launch-On-Capture* et *Launch-On-Last-Shift* sur chaque coeur.

Nous avons présenté dans cette partie la stratégie de test basée sur l'utilisation du micro-testeur. Nous avons pu observer l'impact de l'architecture du micro-testeur sur les wrappers de test et les chaînes de scan du coeur. Des précisions concernant les aspects temporels sur les signaux de commande émis par le micro-testeur ont aussi été abordées. Finalement les performances de ce micro-testeur, en termes de volume de données de test, de temps de test et de surface ont été présentées.

Récapitulons rapidement les points importants de notre approche.

La conception modulaire du micro-testeur

Le micro-testeur est composé de deux types de composants : un Prefetch-Buffer et des TPIUs. Les TPIUs sont les composants qui exécutent les programmes de test. Chaque TPIU est dédié au test d'un coeur. L'ajout d'un nouveau coeur à tester se fait simplement en "branchant" un nouveau TPIU sur le Prefetch-Buffer et en reliant ce TPIU au coeur, par le biais d'un TAM dédié. Le Prefetch-Buffer, sorte de mémoire cache, fournit une interface entre l'interconnect système et les TPIUs.

Cette architecture modulaire, ou les TPIUs sont indépendants les uns des autres permet le test concurrent des différents coeurs. De plus l'accès direct à la mémoire externe (DMA) du Prefetch-Buffer, permet un test rapide du SoC. L'analyse des résultats de temps de test, montre que ce type d'approche est plus rapide que celle choisie dans [Lee05], où le micro-testeur partage un bus de test pour tous les coeurs et attend le processeur embarqué pour la lecture des programmes de test.

De plus, cette architecture modulaire permet d'obtenir une surface sur silicium proportionnelle au nombre de coeurs testés. Les résultats ont montré que cette surface supplémentaire, par coeur, était faible, de l'ordre de 5% supplémentaire pour un coeur de la taille d'un processeur MIPS 32 bits (sans les mémoires caches).

Impact de l'architecture du micro-testeur sur les coeurs testés

L'approche proposée réutilise les composants fonctionnels du système, comme le processeur embarqué ou l'interconnect système, afin de réduire au maximum la taille du micro-testeur. Ainsi, pendant le test de certains coeurs, l'interconnect système est utilisé de façon fonctionnelle. Or, les coeurs testés (et donc munis d'un wrapper IEEE 1500) qui sont connectés à l'interconnect système, doivent pendant leur test, ne pas émettre de valeurs vers l'extérieur (sur l'interconnect système). La norme prévoit et recommande, afin d'isoler le wrapper, d'émettre vers l'extérieur des valeurs "safe". Dans notre cas, cette isolation est obligatoire.

Nous avons aussi vu que l'architecture du micro-testeur ne permettait pas l'envoi des vecteurs de test de façon continue. Le flux de données est envoyé par paquet de 32 bits. Entre deux paquets des cycles d'attente doivent être gérés. Il existe deux façons de gérer ces cycles d'attente : le coeur doit être muni de bascules scan gardant les valeurs pendant ces cycles d'attente, ou mettre en place un contrôle de l'horloge du coeur. Nous avons vu quels étaient les contraintes au niveau temporel de ces deux solutions. L'utilisation de ces nouvelles bascules scan impose que tous les signaux liés au test du coeur puissent êtres positionnés en un cycle d'horloge fonctionnelle. L'utilisation de logique sur l'horloge impose que le signal masquant l'horloge, émis par le micro-testeur, puisse s'activer et se désactiver en une demie période d'horloge fonctionnelle. Dans ce cas, les signaux internes du coeur liés au test, peuvent fonctionner à une fréquence plus basse que la fréquence nominale du coeur.

Fréquence de test approprié au coeur et test at-speed

L'utilisation des nouvelles bascules scan et le contrôle de l'horloge permettent au micro-testeur de gérer la fréquence de fonctionnement du coeur testé.

Cette capacité du micro-testeur est intéressante puisque les signaux liés aux aspects testabilités (comme le signal "scan_enable") ne sont en général pas optimisés, au sens temporel, comme les signaux fonctionnels. C'est pourquoi, il est possible que le test doive s'exécuter avec une fréquence d'horloge plus lente que celle fonctionnelle.

Le micro-testeur peut, de façon logicielle, régler la fréquence d'horloge de test du circuit, suivant les capacités de celui-ci. Si le coeur peut supporter un chargement/déchargement des vecteurs de test à la fréquence nominale du circuit, le micro-testeur peut fournir un scan *at-speed*. Sinon le micro-testeur peut générer une fréquence de scan jusqu'à 32 fois inférieure à la fréquence du système.

La gestion de la fréquence de test des coeurs a aussi permis d'introduire dans le micro-testeur les deux techniques de test *at-speed* gérés par les ATPGs : le *Launch-On-Capture* (LOC) et le *Launch-On-Last-Shift* (LOLS). Ainsi, le micro-testeur peut fournir un test de qualité sur chaque coeur en utilisant les vecteurs générés par les outils de CAO de l'industrie.

La largeur des TAMs

Nous avons vu que le micro-testeur possède un TAM dédié pour chaque coeur testé. L'architecture du micro-testeur rend les performances de celui-ci très sensibles aux largeurs des TAMs utilisés. En effet, le micro-testeur est inséré dans le système comme un composant fonctionnel "classique" et possède donc une architecture structurée sur 32 bits. Or les architectures de test traditionnelles (TAM, wrappers etc.) ne sont pas contraintes à une structure sur 32 bits et sont structurées sur N bits. Le micro-testeur doit faire une conversion entre un flux structuré sur 32 bits et des flux de tailles variables. Ainsi, lorsque les flux en sortie du micro-testeur n'ont pas des tailles diviseurs de 32, des bits de bourrage doivent être insérés dans les programmes de test HTC.

Nous avons pu voir dans les résultats, l'influence de ces bits de bourrage. Lorsque les coeurs testés possèdent un TAM dont la largeur divise 32, aucun bit de bourrage n'est inséré. Au pire, les bits de bourrage peuvent doubler la taille du volume des données de test. Comme nous avons pu le voir, l'augmentation du volume de données impacte directement les temps de test qui sont alors allongés.

Pour résumer, nous avons proposé l'introduction d'un composant dédié au test dans le système : le micro-testeur. Le micro-testeur nous permet de combiner l'approche d'auto-test logiciel (SBST) dans un contexte où les coeurs testés sont compatibles avec la norme IEEE 1500.

Les résultats sur les performances du micro-testeur ont montré que pour une surface supplémentaire faible et une réduction du volume des données test, le test de l'ensemble des coeurs pouvait être plus rapide comparé à un TAM traditionnel piloté par un testeur externe.

Avec cette approche, le test n'est plus bridé par les performances du testeur externe.

Conclusion générale

Avec les progrès accomplis dans le domaine de l'intégration des circuits intégrés, aujourd'hui, les concepteurs sont capables d'intégrer un système complet sur une seule puce (SoC). La conception d'un tel système se fait par assemblage de coeurs (processeurs, DSPs, RAMs, contrôleurs réseaux etc.) autour d'un interconnect (bus, crossbar, NoC etc.), poussant l'industrie à séparer les concepteurs de coeurs d'un côté et les intégrateurs système de l'autre. La diversité des coeurs présents dans un SoC rend leur test extrêmement difficile. En effet, les techniques de DfT actuelles, se limitant à l'usage d'un testeur externe et aux techniques d'auto-test matériel (BIST), augmentent les coûts du test tout en fournissant une qualité de test de moins en moins adaptée aux technologies profondément sub-microniques.

Devant la complexité du test de ces systèmes sur puce, des chercheurs ont essayé de réutiliser la capacité des SoCs à exécuter du logiciel embarqué, à des fins d'auto-test. Cette nouvelle discipline, le *Software-Based Self-Test* (SBST), que nous avons traduit par "auto-test logiciel", permet la réduction des coûts du test, et assure un test de qualité. En effet, le SBST fournit un test au niveau système, minimise les besoins de structures matérielles dédiées au test, tant internes que externes, et permet l'exécution du test à la fréquence nominale du circuit. Nous avons présenté dans ce manuscrit les deux grandes branches de cette discipline. La première branche se focalise sur l'auto-test logiciel des processeurs. Une fois le processeur embarqué testé, il peut être réutilisé pour tester les autres composants du système. Ainsi, la deuxième branche, l'auto-test logiciel des SoCs, se concentre sur le test des autres coeurs présents dans le système.

La contribution de notre travail sur l'auto-test logiciel est double. Dans la première partie, nous avons étudié l'auto-test logiciel des processeurs. Nous avons proposé le développement de routines de test pour les petites mémoires embarqués dans les processeurs telles que les bancs de registres et les mémoires caches. Dans la deuxième partie, notre étude s'est portée sur l'auto-test logiciel des SoCs. Nous avons proposé l'introduction d'un micro-testeur embarqué dans le SoC afin de tester les coeurs munis de wrapper IEEE 1500.

Auto-test logiciel des mémoires embarquées dans les processeurs

Comme nous venons de le préciser, la première partie de ce manuscrit s'est focalisée sur le test des petites mémoires embarquées dans les processeurs. Nous avons vu que, bien que le BIST matériel pour les mémoires soit efficace, il engendre une surface excessive sur les petites mémoires comme

les bancs de registres et les mémoires caches, et dégrade les performances du circuit. Or ces mémoires, sont embarquées dans les coeurs de processeurs qui sont généralement optimisés en surface, en consommation et en fréquence.

Le chapitre 3 de la première partie a étudié l'auto-test logiciel du banc de registres, le chapitre suivant s'est concentré sur l'auto-test logiciel des mémoires caches.

Dans le chapitre concernant l'auto-test logiciel des bancs de registres, nous avons pu constater que les approches proposées ne présentent aucun détail d'implémentation, ni aucun résultat sur les performances des programmes de test. Or, l'approche SBST n'est intéressante que si les programmes de test sont assez légers, afin d'être chargés rapidement par le testeur externe, et s'ils s'exécutent promptement. Une approche fournit des détails sur l'implémentation des programmes de test ainsi que sur les résultats. Malheureusement, cette approche repose sur un banc de registres synthétisé en portes logiques, et ne fournit donc pas de test adapté aux bancs de registres conçus comme des mémoires.

Pour remedier à cela nous avons proposé une méthode systématique permettant la génération automatique de programmes de test, basée sur l'utilisation d'algorithmes March. Nous avons pu constater que les programmes implémentant des algorithmes March simples tel que le MATS, sont légers et rapides à l'exécution. Afin d'effectuer un test de meilleur qualité, l'utilisation d'algorithmes plus complexes, tel que le March C-, est nécessaire, mais conduit à utiliser des programmes plus lourds et moins rapides. Les résultats présentés offrent dorénavant la possibilité d'effectuer un choix d'algorithme en fonction du compromis qualité de test/coût du test.

Le chapitre suivant porte sur l'auto-test logiciel des mémoires caches. Comme pour le banc de registres, nous avons vu que des stratégies avaient été développées mais qu'aucune donnée numérique n'était présentée. Nous avons implémenté des stratégies de test par effet de bord, en se focalisant sur les mémoires caches à correspondance directe et de type "Write-Through". Nous avons pu constater que les programmes de test pour la mémoire située dans le cache de données peuvent être générer automatiquement et sont très légers, alors que ceux pour la mémoire située dans le cache des instructions sont très lourds et difficile à implémenter. Dans les deux cas, nous avons pu voir que les temps de test sont en moyenne six fois plus long qu'une stratégie BIST matérielle partagée. Ces résultats offrent là encore à l'intégrateur système la possibilité de choisir une stratégie ou une autre en fonction des ses contraintes les plus prioritaires.

Auto-test logiciel des SoCs : le micro-testeur embarqué

La deuxième partie de ce manuscrit s'est focalisée sur l'étude de l'auto-test logiciel des SoCs. Un SoC se construit par assemblage de coeurs préconçus tels que les processeurs, les mémoires, les DSPs, ou des blocs dédiés. Afin d'en simplifier le test, le groupe IEEE 1500 a définit un "wrapper" de test à ajouter à chaque coeur.

Au regard de la littérature, le seul moyen de combiner les avantages du SBST tout en gardant la compatibilité IEEE 1500 est d'introduire un composant spécifique dédié au test : le micro-testeur. Connecté d'un côté à l'interconnect système, relié de l'autre au coeurs wrappés IEEE 1500, le micro-testeur permet de faire le lien entre l'approche *fonctionnelle* SBST et le test *structurel* des coeurs.

Nous avons pu constater que les différentes approches, basées sur l'utilisation des micro-testeurs, proposées dans la littérature souffraient de plusieurs défauts. Entre autres, elles ne permettent pas le test concurrent des coeurs, augmentant par conséquent le temps de test. Elles reposent sur la génération pseudo-aléatoire de vecteurs, ou lorsque les vecteurs déterministes fournis avec l'IP sont utilisés et stockés en mémoire externe, le processeur embarqué est alors utilisé pour les récupérer. Le processeur embarqué est en général le goulot d'étranglement de la méthode, aggravant davantage le temps de test. Enfin, et surtout, aucune des approches ne fournit de test *at-speed* structurel de type "Launch-On-Capture" ou "Launch-On-Last-Shift".

Nous avons donc proposé une stratégie basée sur l'utilisation d'un micro-testeur compatible avec la norme IEEE 1500. Les vecteurs de test fournis avec les IPs sont stockés dans une mémoire externe et sont encapsulés dans un format spécifique au micro-testeur. Le processeur embarqué supervise le processus de test au niveau système pendant que le micro-testeur se préoccupe du test des coeurs. Les caractéristiques de notre approche sont :

- Une architecture modulaire du micro-testeur : avoir séparé le module de communication avec l'interconnect système (Prefetch-Buffer) et les modules de test des coeurs (TPIUs), permet au micro-testeur de s'adapter facilement à n'importe quel protocole (de communication ou de test) en instanciant les modules adaptés.
- Un test concurrent des coeurs : les TPIUs sont indépendants les uns des autres et le Prefetch-Buffer agit comme une mémoire cache permettant à chaque TPIU d'exécuter le test des coeurs en parallèle.
- Utilisation des vecteurs déterministes fournis avec l'IP : utiliser ces vecteurs nous permet d'être compatible avec les coeurs fournis sous forme de *hard IP* et permet généralement d'obtenir des taux de couverture meilleurs et plus rapidement qu'avec des vecteurs pseudo-aléatoires calculés par le micro-testeur ou par le processeur embarqué.
- Utilisation d'un DMA : afin d'accélérer le test, le micro-testeur peut accéder directement à la mémoire externe sans avoir recours au processeur embarqué.
- Deux modes de fonctionnement : le micro-testeur peut compacter les réponses obtenues afin d'effectuer le test le plus rapidement possible, c'est le mode test de production. Le micro-testeur a la capacité de comparer la réponse obtenue à la réponse attendue bit par bit, permettant ainsi de déterminer le bit fautif, c'est le mode diagnostic.
- Un test *at-speed* structurel : le micro-testeur est capable d'effectuer un test de type "Launch-On-Capture" et "Launch-On-Last-Shift" sur chacun des coeurs testés.

Les résultats obtenus sur les performances du micro-testeur montrent que pour une surface supplémentaire faible, le micro-testeur permet la réduction du volume des données de test (en mode compaction) et permet d'achever rapidement le test des coeurs. Nous avons comparé les temps de test du micro-testeur à TR-Architect. TR-Architect est le logiciel développé par Philips permettant l'optimisation du TAM dédié TestRail piloté par un testeur externe. Dû au fait que le micro-testeur n'est pas bridé par la fréquence de fonctionnement du testeur externe, contrairement à TR-Architect, les temps de test peuvent être en faveur du micro-testeur.

Cette étude nous a aussi permis d'étudier les incommodités liées à l'utilisation du micro-testeur.

Les stratégies basées sur l'utilisation d'un micro-testeur, réutilise certaines fonctionnalités du SoC pour effectuer le test de certains composants. Ainsi, une partie du système est en mode fonctionnel

pendant qu'une autre est en mode test, à la différence des stratégies par test externe et BIST où la puce bascule complètement en mode test. Les micro-testeurs sont alors à l'interface de ces deux modes. Afin de supporter la transition entre ces deux modes, des contraintes matérielles et temporelles doivent être respectées. Nous avons pu voir les modifications matérielles nécessaires à apporter aux wrappers (effet "Ripple") et aux bascules scan (ajout du mode "Hold"). Nous avons vu les contraintes temporelles (timing) sur les signaux de test dans le cas ou les nouvelles bascules sont utilisées et dans le cas où le micro-testeur contrôle l'horloge du wrapper et du coeur.

Nous avons aussi pu constater la sensibilité du micro-testeur à la largeur du TAM. Pour le moment le micro-testeur limite la largeur des TAMs parallèles à 32 bits maximum. Lorsque la largeur du TAM n'est pas adapté au micro-testeur (largeur non diviseur de 32) l'insertion de bit de bourrage provoque le gonflement des programmes de test et a pour conséquence l'augmentation des temps de test.

Pour conclure, nous pouvons dire que la conversion d'un circuit en circuit testable est d'autant plus facile si l'on considère le test comme une fonctionnalité propre du circuit dès sa conception. Or, aujourd'hui, les fonctionnalités d'un SoC sont conçues de façon conjointe entre matériel et logiciel. La conception en vue du test d'un SoC doit donc être effectuée de façon conjointe entre matériel et logiciel.

Perspectives

Concernant les petites mémoires embarquées dans les processeurs

Cette étude nous a permis de voir dans quelles mesures, l'auto-test logiciel (SBST) des mémoires embarquées dans les microprocesseurs, était une approche plus avantageuse ou moins intéressante qu'une approche d'auto-test matériel (BIST).

En ce qui concerne l'implémentation des programmes de test, cette étude nous a permis d'en tracer les grandes lignes. Néanmoins, ces mémoires ont une conception spécifique, il serait donc intéressant de poursuivre ces études selon des caractéristiques plus fines :

Pour le banc de registres

Le banc de registres est en fait une mémoire multi-ports (3 ports). Le test des mémoires multi-ports requiert des tests spécifiques. En effet, les accès multiples et simultanés peuvent sensibiliser des fautes qui sont différentes des fautes conventionnelles des mémoires simple-port. Typiquement des fautes inter-ports peuvent survenir [Zhao00].

La deuxième particularité est d'avoir un port en écriture seule (write-only) et les deux autres ports en lecture seule (read-only). Ainsi, des algorithmes tenant compte de ces particularités (wo-ro-ro), doivent être implémentés afin d'obtenir une couverture de fautes optimale, en un minimum de temps [Hamdioui98].

Pour les mémoires caches

Notre étude s'est portée sur des caches "simples" à correspondance directe et "Write-Through". Il serait intéressant d'obtenir des résultats sur des caches à associativité différente (partiellement associatif comme les 2 ou 4 voies et totalement associatif), et d'étudier les politiques d'écriture de type "Write-Back".

Finalement, une étude complète et chiffrée, sur les autres composants d'un cache, tel que le propose [Gizopoulos04] pour les coeurs de processeurs, permettrait de proposer une approche intégrale du test des processeurs. Aujourd'hui, la mémoire cache occupe la moitié de la surface sur silicium du processeur.

De plus notre étude s'est limitée au banc de registres et aux mémoires caches, mais les processeurs intègrent de plus en plus de petites mémoires de travail (comme les ROB ReOrder-Buffer) qu'il serait intéressant d'étudier.

Concernant le micro-testeur

Le travail présenté dans la partie concernant le micro-testeur nous a permis d'établir une stratégie de test des coeurs wrappés IEEE 1500.

Nous nous sommes surtout focalisés sur le test internes des coeurs. Nous n'avons pas abordé le test externe, le test des interconnexions entre coeurs. Il est actuellement implémenté un mécanisme de synchronisation entre les TPIUs et le processeur embarqué qui pourrait permettre le test entre les coeurs. Néanmoins, il serait intéressant de fournir un mécanisme de test des interconnexions *at-speed*.

Concernant le mode diagnostic, il faudrait déterminer si le micro-testeur contient une erreur ou non. Une stratégie SBST pourrait être implémentée pour tester le micro-testeur.

Nous avons pu sentir le long de cette étude les différentes contraintes de timing auxquelles les fils de test devaient faire face. Dans ce cadre, il serait intéressant d'étudier au niveau physique (layout) le comportement des TAMs (largeurs et longueurs) entre le micro-testeur et les coeurs. Sachant que le micro-testeur utilise des TAMs dédiés à chaque coeur, il serait intéressant de voir les problèmes de congestion dus au routage des TAMs trop larges. Une étude sur les limites de longueurs des TAMs en fonction de la taille des coeurs testés serait intéressante. Les problèmes de timing pourrait être résolus par des techniques "pipelinisant" les longs fils. Il pourrait être aussi envisagé l'utilisation de plusieurs micro-testeurs physiquement "proches" des coeurs.

Au niveau logiciel, profitant de la puissance de calcul embarquée actuelle, des techniques de génération des vecteurs de test déterministes (pour ne pas perdre la compatibilité IEEE 1500) pourraient être implémentées par les processeurs embarqués ou autres DSPs. Les programmes de test HTC pourraient être générés en interne ce qui permettrait de s'affranchir de la connexion avec la RAM externe.

Pour finir, nous pouvons constater que les possibilités sont vastes, notre stratégie actuelle n'utilise que peu des fonctions embarquées disponibles.

Annexe A

Concernant le banc de registres

Test	DB	Mémoire Parfaite	Tailles des caches instructions et données						Taille du code (en octets)
			1Ko	2Ko	4Ko	8Ko	16Ko	32Ko	
MATS	1	674	2085	2042	2061	2189	2445	2957	2656
	1–6	3944	6915	6272	5556	5684	5940	6452	3136
MATS+	1	921	2842	2814	2773	2901	3157	3669	3644
	1–6	5426	11172	8039	7518	7631	7887	8399	4124
MATS++	1	1058	3234	3206	3180	3293	3549	4061	4192
	1–6	6248	14499	9476	8955	8708	8964	9476	4672
MARCH X	1	1058	3234	3206	3180	3293	3549	4061	4192
	1–6	6248	14499	9476	8955	8708	8964	9476	4672
MARCH C-	1	1826	5442	5474	5478	5501	5757	6269	7264
	1–6	10856	32112	20879	15228	14756	15012	15524	7744

TAB. A.1: Temps de test, en nombre de cycles, dans le cas d'une mémoire parfaite (répondant en 1 cycle) et dans le cas où une mémoire cache est utilisée. La taille de chaque programme est présentée en octets.

Test	DB	SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
MATS	1	100.0 %	100.0 %	59.8 %	69.7 %	69.7 %	37.5 %
	1–6	100.0 %	100.0 %	98.9 %	99.4 %	99.4 %	90.4 %
MATS+	1	100.0 %	100.0 %	96.9 %	69.5 %	80.3 %	44.8 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	99.4 %	92.6 %
MATS++	1	100.0 %	100.0 %	96.9 %	69.5 %	81.5 %	45.4 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	99.4 %	92.8 %
MARCH X	1	100.0 %	100.0 %	96.9 %	73.2 %	100.0 %	55.5 %
	1–6	100.0 %	100.0 %	100.0 %	99.4 %	100.0 %	99.3 %
MARCH C-	1	100.0 %	100.0 %	96.9 %	98.4 %	100.0 %	98.4 %
	1–6	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %

TAB. A.2: Taux de couverture obtenus après simulation par les programmes auto-testants.

Annexe B

Concernant le cache de données

MATS version 1 DB					**MATS** version 6 DBs				
$\#L \setminus \#C$	*2*	*4*	*8*	*16*	$\#L \setminus \#C$	*2*	*4*	*8*	*16*
64	5096	7557	12635	22877	*64*	29186	44314	74934	136342
128	9832	14853	25051	45533	*128*	57282	87770	149110	271958
256	19304	29445	49883	90845	*256*	113474	174682	297462	543190
512	38248	58629	99547	181469	*512*	225858	348506	594166	1085654
MATS+ version 1 DB					**MATS+** version 6 DBs				
$\#L \setminus \#C$	*2*	*4*	*8*	*16*	$\#L \setminus \#C$	*2*	*4*	*8*	*16*
64	5800	8389	13723	24477	*64*	33410	49306	81462	145942
128	11240	16517	27227	48733	*128*	65730	97754	162166	291158
256	22120	32773	54235	97245	*256*	130370	194650	323574	581590
512	43880	65285	108251	194269	*512*	259650	388442	646390	1162454
MATS++ version 1 DB					**MATS++** version 6 DBs				
$\#L \setminus \#C$	*2*	*4*	*8*	*16*	$\#L \setminus \#C$	*2*	*4*	*8*	*16*
64	6330	9435	15801	28619	*64*	36509	55472	93780	170564
128	12282	18587	31353	56971	*128*	71892	110064	186772	340356
256	24186	36891	62457	113675	*256*	142676	219248	372756	679940
512	47994	73499	124665	227083	*512*	284244	437616	744724	1359108
MARCH X version 1 DB					**MARCH X** version 6 DBs				
$\#L \setminus \#C$	*2*	*4*	*8*	*16*	$\#L \setminus \#C$	*2*	*4*	*8*	*16*
64	6712	9796	16159	28970	*64*	38681	57508	95802	172579
128	12984	19268	32031	57642	*128*	75876	114020	190714	344291
256	25528	38212	63775	114986	*256*	150500	227044	380538	687715
512	50616	76100	127263	229674	*512*	299748	453092	760186	1374563
MARCH C- version 1 DB					**MARCH C-** version 6 DBs				
$\#L \setminus \#C$	*2*	*4*	*8*	*16*	$\#L \setminus \#C$	*2*	*4*	*8*	*16*
64	9953	14274	23192	41156	*64*	58685	83896	137538	245053
128	19297	28098	45976	81860	*128*	113073	166520	273922	488957
256	37985	55746	91544	163268	*256*	224561	331768	546690	976765
512	75361	111042	182680	326084	*512*	447537	662264	1092226	1952381

TAB. B.1: Temps de test, en nombre de cycles, en fonction du nombre de colonnes (#C) et du nombre de lignes (#L) du cache de données.

Test		Nombre de colonnes			
		2	*4*	*8*	*16*
MATS	1 DB	280	312	376	504
	6 DBs	584	616	680	808
MATS+	1 DB	284	316	380	508
	6 DBs	588	620	684	812
MATS++	1 DB	300	348	444	636
	6 DBs	604	652	748	940
MARCH X	1 DB	348	396	492	684
	6 DBs	668	716	812	1004
MARCH C-	1 DB	484	564	724	1044
	6 DBs	836	916	1076	1396

TAB. B.2: Tailles des programmes de test (en octets) en fonction du nombre de colonnes du cache.

Test	DB	SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
MATS	1	100.0 %	50.0 %	49.2 %	74.6 %	50.0 %	25.0 %
	1–6	100.0 %	100.0 %	98.5 %	98.5 %	99.2 %	88.3 %
MATS+	1	100.0 %	50.0 %	99.2 %	50.0 %	74.8 %	37.4 %
	1–6	100.0 %	100.0 %	100.0 %	98.4 %	99.2 %	91.0 %
MATS++	1	100.0 %	100.0 %	99.2 %	50.0 %	75.2 %	37.6 %
	1–6	100.0 %	100.0 %	100.0 %	98.4 %	99.2 %	91.1 %
MARCH X	1	100.0 %	100.0 %	99.2 %	62.4 %	100.0 %	50.0 %
	1–6	100.0 %	100.0 %	100.0 %	99.2 %	100.0 %	99.1 %
MARCH C-	1	100.0 %	100.0 %	99.2 %	99.6 %	100.0 %	99.6 %
	1–6	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %

TAB. B.3: Taux de couverture obtenus après simulation par RAMSES des les programmes auto-testants.

Annexe C

Concernant le cache instructions

	#Colonnes			
#Lignes	*2*	*4*	*8*	*16*
64	7,513	9,803	14,328	23,545
128	14,873	19,467	28,600	46,721
256	29,593	38,795	57,144	93,385
512	59,033	77,451	114,232	186,713

TAB. C.1: Temps de test (en nombre de cycles) nécessaires au test March IC en fonction du nombre de colonnes et de lignes du cache instructions.

	#Colonnes			
#Lignes	*2*	*4*	*8*	*16*
32	2,368	4,672	9,280	18,496
64	4,672	9,280	18,496	36,928
128	9,280	18,496	36,928	73,280
256	18,496	36,928	73,792	146,496
512	36,928	73,792	147,520	292,928

TAB. C.2: Tailles des programmes de test (en octets) en fonction du nombre de colonnes et de lignes du cache instructions.

SAF	TF	AF	CF_{st}	CF_{in}	CF_{id}
100.0 %	100.0 %	2.7 %	78.7 %	3.0 %	2.0 %

TAB. C.3: Taux de couverture du test March IC.

Annexe D

Concernant le micro-testeur

D.1 Exemple d'un programme HTC

La figure D.1 présente un exemple de programme de test HTC.

La première instruction `LOADWIR` charge le registre d'instruction du wrapper IEEE 1500 (WIR) du coeur correspondant. Les paramètres sont la taille du WIR et la valeur à charger. Lorsque le TPIU reconnaît l'instruction `LOADWIR` il déclenche alors la séquence d'activation des signaux WSP permettant de charger le WIR. La figure D.2 présente le chronogramme associé à cette séquence de chargement du WIR.

Le rôle de la deuxième instruction est de charger un stimulus de test dans le wrapper. Lorsqu'un vecteur de test contient plus de 32 bits, le vecteur est découpé en tronçons de 32 bits. Si la taille du vecteur n'est pas un multiple de 32, le dernier tronçon est complété avec des bits de bourrage. Ainsi, un paramètre de l'instruction de test `SHIFT_IN` est le nombre de tronçons qui suivent.

La sixième instruction `CAPTURE` active le signal CaptureWR du wrapper afin de capturer les réponses du coeur sous test dans les chaînes de scan et dans le registre de bordure du wrapper.

L'instruction suivante active le mode compaction. Cette instruction n'a qu'un usage interne au TPIU, elle n'affecte pas le wrapper.

Ensuite les réponses capturées sont récupérées hors du wrapper et compactées dans le TPIU (selon l'instruction 7).

Finalement, la dernière instruction `COMPARE`, compare la réponse compactée avec celle attendue (donnée de test 10).

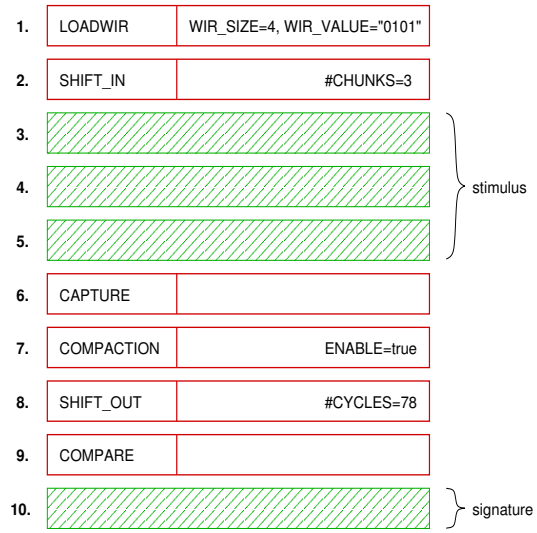


FIG. D.1: Exemple de programme HTC.

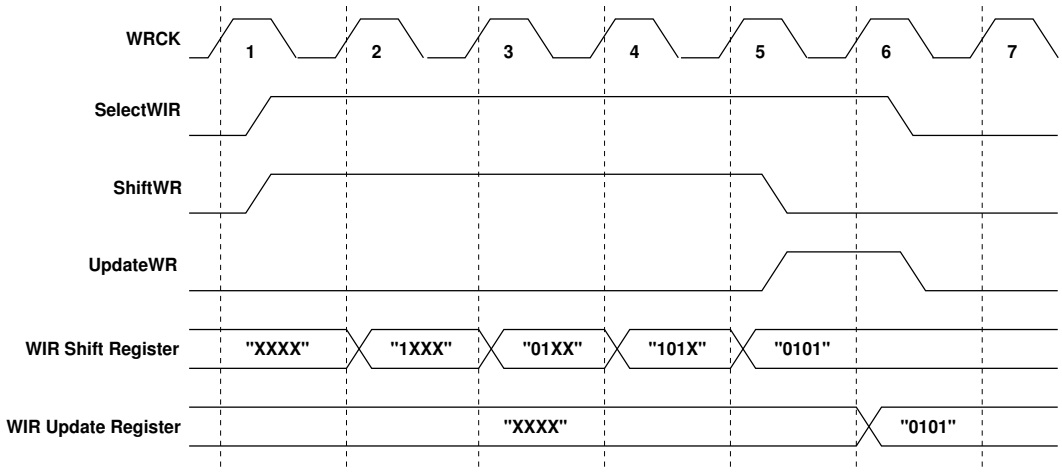


FIG. D.2: Chronogramme de chargement du WIR.

D.2 Jeu d'instruction HTC

Le jeu d'instruction HTC présenté dans le tableau D.1 adresse principalement le test des composants compatibles avec la norme IEEE 1500.

Instruction HTC	Commentaire
Instructions de gestion interne au TPIU	
EOP	dernière instruction du programme
NOP	pas d'opération
INIT_BUF_UNLOAD	initialise le registre Buf_Unload
COMPARE	compare le registre Buf_Unload et les 32 bits de données qui suivent
COMPACTION_ENABLE	active le mode MISR du registre Buf_Unload
COMPACTION_DISABLE	Buf_Unload est un registre à décalage classique
MARKER	place une valeur dans le registre Marker (cf section D.5)
SHIFT_FREQ	sélectionne la fréquence de scan
SERIAL	les instructions suivantes agissent sur le WSP
PARALLEL	les instructions suivantes agissent sur le WPP
MASK_CLK_ENABLE	gestion de la logique sur l'horloge (cf section 9.1.3)
MASK_CLK_DISABLE	pas de gestion de logique sur l'horloge
SYNC	génère une IRQ (permet la synchronisation avec le processeur)
Instructions contrôlant le WSP/WPP	
LOADWIR	charge le WIR du wrapper
LOAD_UNLOAD_WIR	charge le nouveau WIR, décharge l'ancien
UPDATE	active l'évènement Update
CAPTURE	active l'évènement Capture
CAPTURE_UPDATE	active les évènements Update et Capture
SHIFTIN	envoie les vecteurs de test, lorsque le mode compaction est activé, les réponses sont récupérées en même temps
SHIFTOUT	récupère les réponses
LOC_ENABLE	active le mode Launch-On-Capture (cf chapitre 9.3)
LOC_DISABLE	désactive le mode Launch-On-Capture
LOLS_ENABLE	active le mode Launch-On-Last-Shift (cf chapitre 9.3)
LOLS_DISABLE	désactive le mode Launch-On-Last-Shift

TAB. D.1: Les 24 instructions HTC développées.

D.3 La bibliothèque STELA

Les programmes HTC correspondent à l'exécution d'une séquence de mots de 32 bits. Pour écrire facilement ces programmes de test, une bibliothèque appelée STELA, a été développée en langage C++.

Le listing D.1 présente un exemple d'utilisation de la bibliothèque STELA.

La fonction SET_FILE permet de nommer le programme HTC généré. Ensuite, de la ligne 10 à la ligne 12 les fonctions utilisées permettent la configuration du test : taille du WIR, utilisation de l'interface série et désactivation du mode compaction.

De la ligne 15 à la ligne 17 un test d'intégrité du wrapper est effectué. On configure le wrapper pour que seul le WBR soit accessible (ligne 15), et on charge un vecteur de la taille du WBR, ici 10 bits. (ligne 16). Le vecteur est ensuite déchargé et comparé bit par bit avec celui attendu (paramètre de la fonction `UNLOAD`). La fonction `UNLOAD` de la bibliothèque STELA, crée une suite d'instructions HTC `SHIFT_OUT/COMPARE`.

Listing D.1: Programme de test utilisant la bibliothèque STELA

```
1   #include "stela.hh"
2
3   #define WS_INTEST_RING   "0101"
4   #define WS_INTEST_SCAN   "0100"
5
6   int main( void )
7   {
8      SET_FILE( "test_program0.htc" );
9
10     SET_WIR_SIZE( 4 );
11     USE_SIL();
12     SET_COMPACTION( false );
13
14     // verifie l'integrite du wrapper
15     LOAD_WIR( WS_INTEST_RING ); // verifie le WBR
16     LOAD  ( wbr_integrity );    // wbr_integrity = "0101010101"
17     UNLOAD( wbr_integrity );
18
19     LOAD_WIR( WS_INTEST_SCAN ); // verifie le WBR + chaine de scan
20     LOAD  ( wdr_integrity );    // wdr_integrity = "01010101010101"
21     UNLOAD( wdr_integrity );
22
23     // test du coeur
24     for ( int i = 0; i < 7; i++ )
25     {
26        LOAD( stimuli[i] );
27        CAPTURE();
28        UNLOAD( responses[i] );
29     }
30
31  }
```

De la ligne 19 à 21, le même test est effectué sur l'ensemble WBR concaténé avec une chaîne de scan. Dans l'exemple la taille de la chaîne de scan est de 4 bits.

De la ligne 24 à 28 le test du coeur à proprement parlé est effectué. On suppose qu'il existe un tableau contenant les 8 stimuli nécessaires et un deuxième contenant les 8 réponses attendues. A la ligne 26 un stimulus est chargé dans le wrapper. Une fois chargé, les réponses sont capturées (ligne 27) et sont déchargées pour être comparées bit par bit. Cette opération est répétée sur l'ensemble des 8 vecteurs de test du coeur.

Un script appelé **GenSTELA** permet de passer du programme de test écrit en C++ au programme de test exécutable par le micro-testeur. La figure D.3 dévoile ce flot d'exécution.

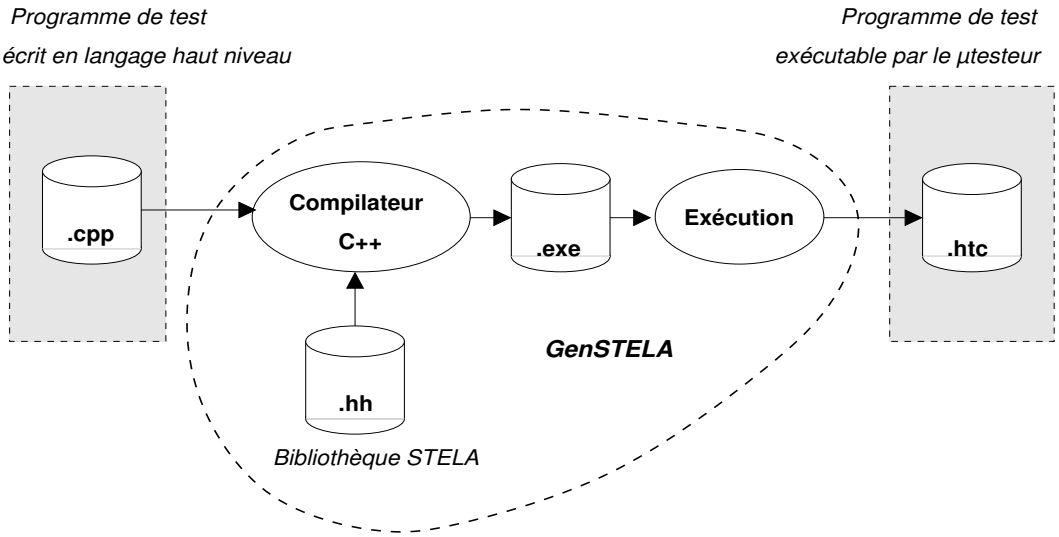


FIG. D.3: Chronogramme de chargement du WIR.

D.4 Le programme de test maître

Le programme de test maître exécuté par le processeur supervise l'exécution des programmes HTC. Ce programme maître peut être écrit en assembleur ou dans des langages plus haut niveau comme le langage C. Ensuite le programme est (cross-)compilé et le binaire résultant est chargé dans la mémoire interne du SoC ou stocké dans la mémoire externe. Ecrire dans des langage haut niveau permet de concevoir un programme de test maître aussi complexe que désiré : plan de test intelligent, redémarrage des programmes de test sur des parties spécifiques, ou même génération des programmes HTC à la volée. En effet, la génération des stimuli peut être effectuée en utilisant des techniques décrites dans les articles [Huang01, Iyer02]. Cependant, le programme devrait implémenter au moins l'algorithme minimal suivant :

Listing D.2: Version minimal d'un programme de test maître

```
1  test_finished = 0;
2
3  for tpiu_x in TPIU_NUMBER {
4     start_test( tpiu_x );
5  }
6
7  while( test_finished != TPIU_NUMBER ){
8      wait( test_interruption );
9
10     // recuperer le numero du TPIU interrompu
11     tpiu_x = get_ICU_interruption();
12     clear_TPIU_interruption( tpiu_x );
13
14     // collecter plus ou moins d'information selon
15     // le mode selectionne: Go/No-Go ou Diagnostic
16     get_informations( tpiu_x );
17     store_informations_outside();
18
19     test_finished = test_finished+1;
20 }
21
22 store_informations_outside( TEST_FINISH );
23 exit();
```

D.5 Le processus de diagnostic

D.5.1 Génération du fichier FDR à l'aide du micro-testeur

Afin de ne pas avoir recours à un ATE pour cette phase de diagnostic, nous avons doté le micro-testeur de certaines capacités afin de pouvoir récupérer les informations nécessaires au diagnostic. Comme nous l'avons vu, l'ATE crée un fichier FDR contenant les valeurs observées ne correspondant pas à celles attendues. Le processus de diagnostic que nous avons développé avec le micro-testeur a donc pour but de récupérer les informations nécessaires à la génération du fichier FDR.

Nous pouvons signaler le fait que le test avec le micro-testeur, de la même manière qu'un BIST, nous permet directement de savoir quel coeur est fautif.

Description du fichier FDR

Les informations contenues dans le fichier FDR sont de la forme :

```
pattern_n  port  [cell_pos] [(expected_data/measured_data)]
```

Le premier item *pattern_n* correspond au numéro de pattern pour lequel une erreur a été observée. Ce numéro correspond au numéro de pattern dans le fichier STIL/CTL.

L'item suivant est le nom du port où l'erreur a été détectée.

L'item *cell_pos* doit être fourni dans le cas où l'erreur est observée dans un chaîne de scan. Il correspond à la position de la cellule fautive dans la chaine de scan. La position de la cellule la plus proche de la sortie est 0, la suivante est 1 et ainsi de suite.

Le dernier item est facultatif.

Dans le cas de coeurs wrappés, seuls les défauts intervenant dans une chaîne de scan sont à considérer. Ainsi les informations à fournir sont donc : le numéro de pattern fautif, le nom de la chaine de scan et enfin le numéro de la cellule fautive. Pour le moment, l'implémentation du mode diagnostic ne gère seulement que l'accès série du wrapper (i.e. le Wrapper Serial Port). Dans ce cas précis, le nom du port scan-out est connu : Wrapper Serial Output (WSO).

Pour résumer, seules deux informations doivent être récupérées : le numéro du pattern fautif et la position de la cellule de la chaine de scan qui a capturé la donnée erronée.

Dans le mode diagnostic, le programme HTC ne contient pas de signature mais toutes les réponses attendues bit par bit. Le compacteur du TPIU sélectionné est désactivé et agit comme un registre à décalage classique où les réponses données par le coeur testé sont comparées bit à bit avec celles attendues.

Le mode diagnostic développé doit être effectué en deux phases. La première phase permet d'obtenir le numéro du pattern fautif. La deuxième phase a pour but d'obtenir la position de la cellule interne de la chaîne de scan.

Phase 1 : identification des patterns fautifs

Le numéro du pattern fautif ainsi que le numéro de la cellule concernée sont obtenus grâce à un registre interne spécifique dans le TPIU appelé MARKER. Dans le première phase, avant l'exécution de chaque pattern de test, qui consiste en l'envoi des stimuli de test, la capture des réponses, et la récupération des réponses, nous enregistrons grâce à une instruction HTC spécifique le numéro du pattern en cours de traitement dans le registre MARKER. Ainsi, si le pattern est fautif, une comparaison erronée lèvera l'IRQ du TPIU. Le processeur est alerté et peut récupérer la valeur rangée dans le registre MARKER du TPIU appelant. Le numéro du pattern fautif est alors récupéré.

La figure D.4 présente le programme STELA permettant de trouver le pattern fautif. Avant le traitement de chaque pattern, son numéro est stocké dans le registre MARKER. La figure D.5 montre le programme HTC correspondant. Dans cet exemple la comparaison avec la valeur attendue pour le pattern numéro 58 échoue et déclenche une interruption.

Phase 2 : identification des cellules fautives

Après cette première phase, nous avons le numéro de tous les patterns fautifs. Dans la seconde phase un schéma quasiment identique est appliqué pour récupérer la cellule fautive. Dans cette phase, le registre MARKER va servir à marquer chaque bit comparé. L'algorithme se présente comme suit : mettre un MARKER i, décaler d'un bit, comparer, mettre un MARKER $i + 1$, décaler d'un bit,

```
1       ...
2   for ( int i=0; i<712; i++ )
3   {
4    SET_MARKER( i );
5    LOAD( stimuli[i] );
6    CAPTURE();
7    UNLOAD( responses[i] );
8   }
9       ...
```

FIG. D.4: Diagnostic phase 1.

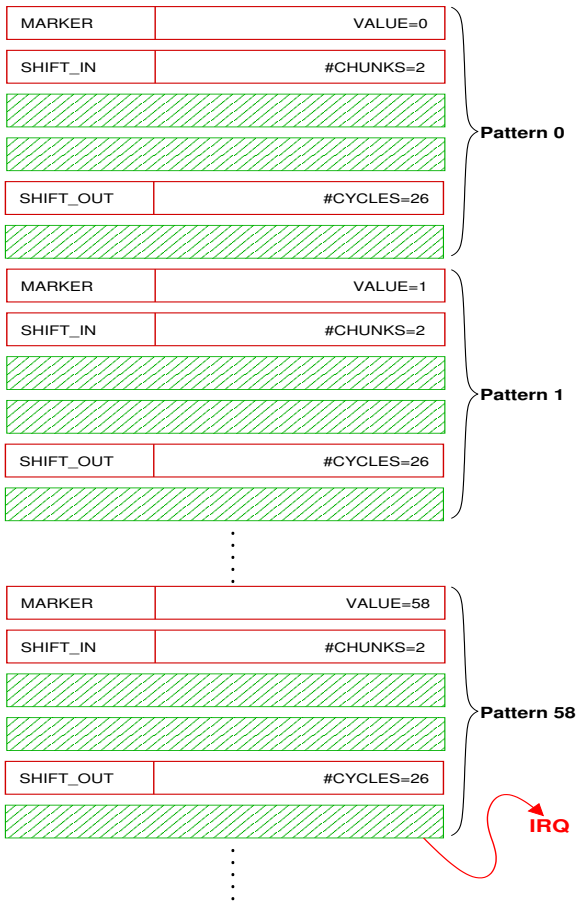


FIG. D.5: Programme HTC correspondant.

comparer, etc. Lorsqu'une comparaison échoue, l'IRQ est activée. Le processeur peut alors ramener la valeur de la cellule fautive stockée dans le registre MARKER. Une fonction STELA a été créée afin d'effectuer cette opération : `UNLOAD_DIAG`. La figure D.6 présente l'utilisation de cette fonction. La figure D.7 montre le programme HTC généré par la fonction `UNLOAD_DIAG`. Les lignes 3, 6, 9 et 33 de la figure D.7 montrent la réponse attendue décalée bit par bit. Les bits attendus sont affichés de façon alphabétique afin de bien voir le décalage. Dans cet exemple, la comparaison en ligne 33 déclenche une interruption. La récupération de la valeur du MARKER (ayant la valeur 10), nous indique que la valeur attendue 'k' est différente dans le TPIU.

Toutes ces étapes du diagnostic sont implémentées en logiciel afin d'alléger le côté matériel du micro-testeur et par là permet de réduire la surface additionnelle.

La taille du MARKER est de 16 bits permettant en phase 1 de différencier 65536 patterns, et en phase 2 de pouvoir sonder une chaîne de scan de 65536 cellules.

```
1       ...
2   int pf = 3; // pattern fautif
3   LOAD( stimuli[pf] );
4   CAPTURE();
5   UNLOAD_DIAG( responses[pf] );
6       ...
```

FIG. D.6: Diagnostic phase 2.

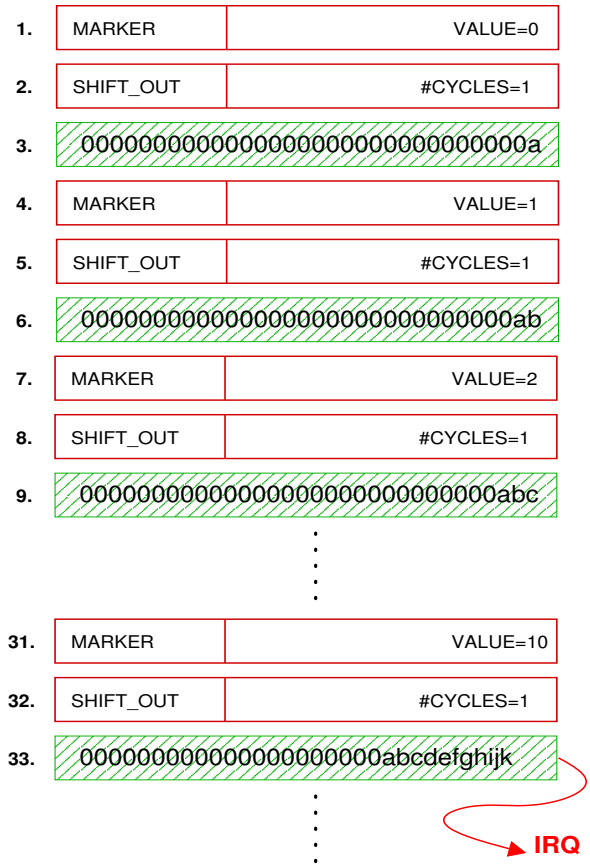


FIG. D.7: HTC généré par la fonction UNLOAD_DIAG().

.

D.6 Résultats

		Mode d'accès aux coeurs					
		mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
$\frac{f_{sys}}{f_{scan}}$	Tailles des FIFOs	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*
1	*64*	364 498	198 876	51 939	27 643	48 859	25 910
	32	364 460	198 834	58 079	30 270	55 229	28 972
	16	364 522	198 900	69 335	36 411	67 989	35 176
	8	364 890	199 076	94 481	48 675	93 503	47 978
	4	371 794	201 810	146 276	74 357	145 562	72 784
2	*64*	715 952	390 754	66 682	35 574	52 524	28 020
	32	715 908	390 710	69 943	37 424	57 672	30 216
	16	715 976	390 778	79 658	41 690	68 943	35 009
	8	716 054	390 820	97 534	51 306	94 724	48 194
	4	717 754	391 634	147 759	75 843	145 562	73 442
4	*64*	1 418 860	774 510	102 710	63 849	65 958	35 263
	32	1 418 798	774 448	104 434	63 889	68 740	36 449
	16	1 418 880	774 530	110 082	64 249	76 800	40 283
	8	1 418 912	774 550	124 138	65 813	97 260	50 722
	4	1 419 576	774 772	166 966	86 153	147 496	74 483
8	*64*	2 824 676	1 542 022	191 234	125 259	105 012	57 033
	32	2 824 614	1 541 960	191 206	125 245	106 452	58 475
	16	2 824 680	1 542 026	191 548	125 335	110 356	60 470
	8	2 824 672	1 542 018	196 974	125 557	120 922	64 799
	4	2 825 076	1 542 160	222 118	128 155	161 366	84 358
16	*64*	5 636 308	3 077 046	371 683	248 203	193 782	106 398
	32	5 636 246	3 076 984	371 669	248 189	193 790	106 296
	16	5 636 284	3 077 022	371 731	248 251	194 128	106 508
	8	5 636 272	3 077 010	371 763	248 277	197 300	108 050
	4	5 636 590	3 077 110	379 878	248 965	218 082	117 556
32	*64*	11 259 572	6 147 094	740 835	494 091	373 326	205 718
	32	11 259 510	6 147 032	740 821	494 077	373 266	205 656
	16	11 259 494	6 147 016	740 825	494 081	373 248	205 640
	8	11 259 518	6 147 040	740 855	494 117	373 508	205 708
	4	11 259 782	6 147 124	741 409	494 287	378 992	208 190

TAB. D.2: Temps de test, en nombre de cycles, pour effectuer le test du benchmark D695, en fonction du mode d'accès aux coeurs, de la taille des FIFOs dans le Prefetch-Buffer et de la fréquence de scan.

		Mode d'accès aux coeurs					
		mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
$\frac{f_{sys}}{f_{scan}}$	Tailles des FIFOs	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*
1	*64*	275 460	155 781	83 060	49 626	45 798	27 649
	32	275 264	155 627	84 762	51 233	50 526	29 919
	16	275 424	155 863	91 288	54 453	60 142	34 674
	8	277 352	157 659	105 316	63 750	80 946	45 124
	4	293 316	168 542	143 806	83 809	124 848	70 504
2	*64*	536 560	301 271	147 441	86 071	49 678	31 467
	32	536 352	301 063	147 423	86 227	54 356	33 691
	16	536 414	301 123	148 490	87 191	63 968	38 380
	8	536 570	301 284	155 810	91 352	83 246	48 050
	4	541 940	305 106	181 399	107 972	127 122	73 140
4	*64*	1 058 808	592 321	280 569	161 901	62 606	42 391
	32	1 058 600	592 112	280 361	161 693	65 360	43 587
	16	1 058 654	592 166	280 413	161 757	72 444	46 760
	8	1 058 680	592 223	281 549	162 591	91 830	56 076
	4	1 061 848	593 733	295 232	172 097	131 742	76 950
8	*64*	2 103 304	1 174 421	546 825	313 581	99 386	68 948
	32	2 103 096	1 174 213	546 617	313 373	100 532	69 473
	16	2 103 134	1 174 249	546 653	313 409	103 756	70 986
	8	2 103 160	1 174 273	546 839	313 539	114 746	75 900
	4	2 105 560	1 175 075	553 317	317 247	149 642	93 152
16	*64*	4 192 296	2 338 621	1 079 337	616 941	178 404	125 152
	32	4 192 088	2 338 413	1 079 129	616 733	178 866	125 186
	16	4 192 094	2 338 417	1 079 133	616 737	180 402	125 764
	8	4 192 130	2 338 457	1 079 211	616 809	185 432	127 996
	4	4 194 206	2 338 851	1 082 649	618 245	208 850	139 462
32	*64*	8 370 280	4 667 021	2 144 361	1 223 661	342 120	239 832
	32	8 370 072	4 666 813	2 144 153	1 223 453	341 912	239 612
	16	8 370 014	4 666 753	2 144 093	1 223 393	341 886	239 558
	8	8 370 068	4 666 809	2 144 171	1 223 457	342 850	239 890
	4	8 371 860	4 667 067	2 146 673	1 224 099	352 976	244 507

TAB. D.3: Temps de test, en nombre de cycles, pour effectuer le test du benchmark G1023, en fonction du mode d'accès aux coeurs, de la taille des FIFOs dans le Prefetch-Buffer et de la fréquence de scan.

		Mode d'accès aux coeurs					
		mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
$\frac{f_{sys}}{f_{scan}}$	Tailles des FIFOs	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*
1	*64*	4 349 394	2 192 215	1 215 528	758 129	613 017	327 569
	32	4 350 536	2 193 797	1 257 290	799 166	679 473	363 107
	16	4 356 650	2 199 625	1 342 472	846 566	812 651	436 371
	8	4 399 158	2 232 910	1 584 986	1 017 910	1 093 887	587 527
	4	4 568 564	2 330 109	2 293 017	1 429 618	1 700 011	912 063
2	*64*	8 558 398	4 311 299	2 156 455	1 331 609	680 395	361 359
	32	8 558 048	4 310 950	2 182 413	1 362 293	742 647	396 759
	16	8 558 372	4 311 242	2 204 976	1 391 410	871 301	464 647
	8	8 565 316	4 317 028	2 378 406	1 492 680	1 139 611	609 445
	4	8 616 636	4 352 748	2 862 070	1 824 696	1 729 109	927 155
4	*64*	16 980 698	8 552 198	4 101 738	2 508 506	861 011	453 895
	32	16 980 298	8 551 798	4 115 485	2 514 534	911 079	481 401
	16	16 980 438	8 551 944	4 150 764	2 537 951	1 021 499	540 109
	8	16 980 874	8 552 342	4 245 959	2 589 909	1 264 683	672 013
	4	16 993 970	8 560 952	4 601 237	2 810 339	1 796 291	960 783
8	*64*	33 825 298	17 033 998	8 075 550	4 908 912	1 279 557	662 575
	32	33 824 898	17 033 598	8 076 830	4 909 806	1 317 791	686 181
	16	33 825 024	17 033 726	8 085 757	4 916 786	1 407 023	736 235
	8	33 825 078	17 033 808	8 129 572	4 948 621	1 603 485	844 529
	4	33 828 904	17 036 064	8 344 017	5 058 791	2 073 667	1 112 635
16	*64*	67 514 498	33 997 598	16 061 182	9 739 486	2 261 475	1 145 521
	32	67 514 098	33 997 198	16 061 166	9 739 473	2 269 497	1 150 459
	16	67 514 192	33 997 294	16 061 588	9 739 821	2 310 161	1 174 467
	8	67 514 228	33 997 338	16 066 992	9 744 349	2 440 077	1 259 350
	4	67 515 864	33 998 264	16 158 134	9 786 311	2 798 589	1 476 673
32	*64*	134 892 898	67 924 798	32 033 150	19 402 077	4 358 867	2 207 869
	32	134 892 498	67 924 398	32 033 134	19 402 061	4 359 631	2 207 925
	16	134 892 528	67 924 430	32 033 358	19 402 299	4 364 455	2 210 613
	8	134 892 578	67 924 466	32 034 024	19 402 793	4 394 575	2 227 094
	4	134 893 840	67 924 936	32 082 866	19 418 959	4 576 855	2 332 259

TAB. D.4: Temps de test, en nombre de cycles, pour effectuer le test du benchmark P22810, en fonction du mode d'accès aux coeurs, de la taille des FIFOs dans le Prefetch-Buffer et de la fréquence de scan.

		Mode d'accès aux coeurs					
		mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
$\frac{f_{sys}}{f_{scan}}$	Tailles des FIFOs	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*
1	*64*	10 382 005	5 219 229	2 233 025	1 267 241	1 252 405	704 317
	32	10 387 615	5 229 149	2 524 357	1 436 175	1 414 647	795 507
	16	10 412 309	5 247 119	3 135 167	1 778 395	1 739 347	978 203
	8	10 486 179	5 316 489	4 318 947	2 450 825	2 394 973	1 346 797
	4	10 704 143	5 545 357	6 717 073	3 810 901	3 724 257	2 093 711
2	*64*	20 432 511	10 261 147	3 084 719	1 708 123	1 380 785	769 899
	32	20 433 107	10 261 737	3 127 897	1 731 969	1 495 465	837 317
	16	20 435 953	10 265 687	3 760 477	2 038 957	1 798 973	1 009 439
	8	20 452 247	10 284 843	4 386 079	2 484 931	2 414 193	1 356 659
	4	20 550 717	10 394 011	6 774 059	3 840 053	3 736 369	2 100 339
4	*64*	40 544 191	20 357 775	4 899 267	3 236 951	1 852 445	979 433
	32	40 543 923	20 357 533	4 912 715	3 243 680	1 930 279	1 037 253
	16	40 544 057	20 357 943	5 030 479	3 274 393	2 096 137	1 154 571
	8	40 546 301	20 361 317	5 905 219	3 383 992	2 560 753	1 446 767
	4	40 574 385	20 396 637	7 796 599	4 238 983	3 837 083	2 153 293
8	*64*	80 768 247	40 553 351	9 244 521	6 394 176	2 947 485	1 485 423
	32	80 767 979	40 553 083	9 244 297	6 392 707	2 996 391	1 516 795
	16	80 768 057	40 553 161	9 245 681	6 395 436	3 120 779	1 603 231
	8	80 768 175	40 553 261	9 358 281	6 424 774	3 416 301	1 821 193
	4	80 778 723	40 564 885	10 547 681	6 607 652	4 429 237	2 462 243
16	*64*	161 216 359	80 944 503	17 940 193	12 710 426	5 385 477	2 707 783
	32	161 216 091	80 944 235	17 939 925	12 708 925	5 402 899	2 711 763
	16	161 216 141	80 944 285	17 939 975	12 709 103	5 445 075	2 735 457
	8	161 216 241	80 944 363	17 951 517	12 714 306	5 609 047	2 833 617
	4	161 221 315	80 950 413	18 270 101	12 793 910	6 132 141	3 195 297
32	*64*	322 112 583	161 726 807	35 331 537	25 342 938	10 428 801	5 247 149
	32	322 112 315	161 726 539	35 331 269	25 341 437	10 428 357	5 246 731
	16	322 112 301	161 726 525	35 331 255	25 341 531	10 431 829	5 247 195
	8	322 112 415	161 726 635	35 335 897	25 343 602	10 468 763	5 259 527
	4	322 116 733	161 728 437	35 598 115	25 379 162	10 718 085	5 423 937

TAB. D.5: Temps de test, en nombre de cycles, pour effectuer le test du benchmark P34392, en fonction du mode d'accès aux coeurs, de la taille des FIFOs dans le Prefetch-Buffer et de la fréquence de scan.

		Mode d'accès aux coeurs					
		mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
$\frac{f_{sys}}{f_{scan}}$	Tailles des FIFOs	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*
1	*64*	10 904 289	5 492 736	2 948 167	1 525 367	2 164 131	1 125 619
	32	10 910 015	5 496 735	3 304 201	1 709 747	2 420 093	1 259 093
	16	10 926 956	5 514 053	4 016 841	2 079 133	2 932 521	1 526 713
	8	11 069 874	5 606 886	5 480 823	2 837 089	3 990 773	2 078 425
	4	12 070 987	6 198 144	8 523 757	4 410 903	6 205 963	3 230 857
2	*64*	21 465 005	10 804 616	4 190 227	2 308 057	2 307 897	1 198 417
	32	21 465 116	10 805 061	4 286 824	2 330 437	2 558 397	1 328 801
	16	21 467 648	10 807 021	4 470 043	2 491 405	3 068 671	1 594 891
	8	21 481 129	10 819 539	5 595 965	2 888 712	4 089 547	2 128 115
	4	21 710 949	10 969 372	8 600 517	4 450 283	6 271 933	3 264 225
4	*64*	42 594 919	21 438 403	7 914 550	4 315 454	2 633 163	1 362 467
	32	42 594 899	21 438 349	7 962 747	4 327 503	2 864 425	1 482 999
	16	42 594 885	21 438 369	8 055 943	4 412 356	3 359 125	1 742 233
	8	42 596 629	21 440 620	8 312 197	4 586 991	4 364 985	2 265 985
	4	42 632 993	21 470 185	9 334 967	5 035 607	6 412 649	3 336 215
8	*64*	84 855 543	42 707 315	15 575 102	8 428 171	3 496 091	1 796 498
	32	84 855 467	42 707 239	15 608 313	8 436 780	3 700 289	1 904 285
	16	84 855 471	42 707 243	15 656 817	8 471 045	4 119 361	2 125 227
	8	84 855 883	42 707 597	15 824 103	8 554 225	5 017 541	2 597 731
	4	84 863 394	42 713 429	16 378 047	8 847 705	6 985 965	3 628 933
16	*64*	169 376 791	85 245 139	30 969 257	16 726 341	5 674 769	2 866 932
	32	169 376 715	85 245 063	30 970 308	16 727 152	5 731 813	2 900 746
	16	169 376 687	85 245 035	30 993 469	16 747 320	5 994 734	3 055 805
	8	169 376 943	85 245 309	31 089 565	16 800 694	6 703 278	3 441 283
	4	169 379 639	85 246 871	31 351 008	16 943 769	8 467 107	4 375 283
32	*64*	338 419 287	170 320 787	61 849 401	33 394 129	10 911 970	5 503 513
	32	338 419 211	170 320 711	61 849 339	33 394 069	10 915 293	5 505 031
	16	338 419 459	170 320 959	61 849 671	33 394 463	10 932 264	5 513 741
	8	338 419 295	170 320 795	61 854 562	33 398 261	11 073 838	5 598 363
	4	338 420 461	170 321 591	61 991 148	33 476 358	12 053 693	6 137 590

TAB. D.6: Temps de test, en nombre de cycles, pour effectuer le test du benchmark P93791, en fonction du mode d'accès aux coeurs, de la taille des FIFOs dans le Prefetch-Buffer et de la fréquence de scan.

benchmark	Volume des données de test brutes		
	Volume des vecteurs de test à charger (Inputs + scan chains)	Volume des vecteurs de test à comparer (Outputs + scan chains)	Total
d695	584 331	642 281	1 226 612
g1023	453 997	451 034	905 031
p22810	6 660 364	6 295 773	12 956 137
p34392	14 761 949	13 363 531	28 125 480
p93791	27 866 043	26 634 760	54 500 803

TAB. D.7: Volume des données de test brutes en bits, pour chaque benchmark.

benchmark	*Mode d'accès aux coeurs*					
	mode série		*mode parallèle (chaînes de scan fixes)*		*mode parallèle (chaînes de scan flexibles)*	
	compaction activée	*compaction désactivée*	*compaction activée*	*compaction désactivée*	*compaction activée*	*compaction désactivée*
d695	659 200	1 311 776	659 200	1 318 304	651 200	1 311 776
g1023	651 680	1 122 240	668 832	1 152 256	613 504	1 122 240
p22810	9 178 304	15 515 232	11 762 048	20 938 624	8 307 232	15 515 232
p34392	21 131 392	34 030 112	34 816 832	61 393 664	19 120 032	34 030 112
p93791	30 037 408	56 709 920	40 288 000	77 902 400	29 500 896	56 709 920

TAB. D.8: Volume total des programmes HTC en bits, pour chaque benchmark, en fonction du mode d'accès aux coeurs et du mode compaction.

Bibliographie

[Abraham85] J. Abraham et H. Shih. Testing of MOS VLSI Circuits. Dans *Proceedings International Symposium on Circuits and Systems (ISCAS)*, pages 1297–1300. 1985.

[ALL] ALLIANCE CAD tools Web Site. http ://www-asim.lip6.fr/recherche/alliance/.

[Amory06] A. M. Amory, K. Goosens et E. Marinissen. Wrapper Design for the Reuse of Networks-on-Chip as Test Access Mechanism. Dans *Proceedings IEEE European Test Symposium (ETS)*, pages 213–218. 2006.

[Batcher99] K. Batcher et C. Papachristou. Instruction Randomization Self Test for Processor Cores. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 34–40. Dana Point, CA, avril 1999.

[Beck05] M. Beck, O. Barondeau, M. Kaibel, F. Poehl, X. Lin et R. Press. Logic Design for On-Chip Test Clock Generation - Implementation Details and Impact on Delay Test Quality. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 56–61. 2005.

[Benabdenbi00] M. Benabdenbi, W. Maroufi et M. Marzouki. CAS-BUS : A Scalable and Reconfigurable Test Access Mechanism for Systems on a Chip. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 141–145. Paris, France, mars 2000.

[Bennetts06] B. Bennetts et B. Kruseman. Notes Tutorial 2 – Delay-Fault Testing : From Basics to ASICs. Dans *Proceedings IEEE European Test Symposium (ETS)*. Southampton, U.K., 2006.

[Bernardi04] P. Bernardi, M. Rebaudengo et M. S. Reorda. Using Infrastructure IPs to support SW-based Self-Test of Processor Cores. Dans *5th International Workshop on Microprocessor Test and Verification (MTV)*, pages 22–27. 2004.

[Bhavsar94] D. Bhavsar et J. Edmondson. Testability Strategy of the Alpha AXP 21164 Microprocessor. Dans *Proceedings IEEE International Test Conference (ITC)*. 1994.

[Brahme84] D. Brahme et J. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, 33(6) : pages 475–485, 1984.

[Bushnell00] M. L. Bushnell et V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI Circuits*. Kluwer Academic Publishers, 2000.

[CADENCE] CADENCE. Cadence Web Site. http ://www.cadence.com/.

[Carbine97] A. Carbine et D. Feltham. Pentium Pro processor design for test and debug. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 294–303. 1997.

[Chen99] L. Chen et S. Dey. A Deterministic Functional Self-Test Methodology for Processors. Dans *Proceedings IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 17–22. November 1999.

[Chen00a] L. Chen et S. Dey. DEFUSE : A Deterministic Functional Self-Test Methodology for Processors. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 255–262. April 2000.

[Chen00b] L. Chen *et al.*. Embedded Hardware and Software Self-Testing Methodologies for Processor Cores. Dans *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 625–630. Los Angeles, CA, juin 2000.

[Chen01a] L. Chen, X. Bai et S. Dey. Testing for interconnect crosstalk defects using on-chip embedded processor cores. Dans *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 317–320. 2001.

[Chen01b] L. Chen et S. Dey. Software-Based Self Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Design*, mars 2001.

[Chen02] L. Chen, X. Bai et S. Dey. Testing for Interconnect Crosstalk Defects Using On-Chip Embedded Processor Cores. *Journal of Electronic Testing : Theory and Applications*, 18(4/5) : pages 165–174, August/October 2002.

[Chen03] L. Chen, S. Ravi, A. Raghunathan et S. Dey. A scalable software-based self-test methodology for programmable processors. Dans *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 548–553. June 2003.

[Corno01a] F. Corno, G. Cumani, M. S. Reorda et G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. Munich, Germany, mars 2001.

[Corno01b] F. Corno, M. S. Reorda, G. Squillero et M. Violante. On the Test of Microprocessor IP Cores. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. Munich, Germany, mars 2001.

[Cota01] E. Cota, F. Brisolara et L. Carro. MET : An Embedded Processor for Test Controlling. Dans *Digest of Papers of IEEE International Workshop on Testing Embedded Core-Based Systems (TECS)*. 2001.

[Cota03] E. Cota, M. Kreutz, C. Zeferino, L. Carro, M. Lubaszewski et A. Susin. The impact of NoC reuse on the testing of core-based systems. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 128–133. 2003.

[DaSilva03] F. DaSilva, Y. Zorian, L. Whetsel, K. Arabi et R. Kapur. Overview of the IEEE P1500 Standard. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 988–997. 2003.

[Dekker88a] R. Dekker, F. Beenker et L. Thijssen. A Realistic Self-Test Machine for Static Random Access Memories. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 353–361. 1988.

[Dekker88b] R. Dekker, F. Beenker et L. Thijssen. Fault Modeling and Test Algorithm Development for Static Random Access Memories. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 343–352. 1988.

[Dekker90] R. Dekker, F. Beenker et L. Thijssen. A Realistic Fault Model and Test Algorithms for Static Random Access Memories. Dans *IEEE Transactions on Computer-Aided Design*, tome 9(6), pages 567–572. IEEE Press, June 1990.

[Feige98] C. Feige, J. ten Pierick, C. Wouters, R. Tangelder et H. Kerkhoff. Integration of the Scan-Test Method into an Architecture Specific Core-Test Approach. Dans *Digest of Papers of IEEE European Test Workshop (ETW)*. Barcelona, Spain, mai 1998.

[Garcia06] O. Garcia. Auto-test logiciel du cache AOC. Rapport technique, LIP6, septembre 2006.

[Gelsinger00] P. Gelsinger. Discontinuities Driven by a Billion Connected Machines. *IEEE Design & Test of Computers*, 17(1) : pages 7–15, 2000.

[Girard02] P. Girard. Survey of Low-Power Testing of VLSI Circuits. *IEEE Design & Test of Computers*, 19(3) : pages 82–92, 2002.

[Gizopoulos04] D. Gizopoulos, A. Paschalis et Y. Zorian. *Embedded Processor-Based Self-Test*. Springer, 2004.

[Goel80] P. Goel. Test Generation Costs Analysis and Projections. Dans *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 77–84. 1980.

[Goel02] S. K. Goel et E. J. Marinissen. Effective and Efficient Test Architecture Design for SOCs. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 529–538. Baltimore, MD, octobre 2002.

[Goor91] A. J. van de Goor. *Testing Semiconductor Memories : Theory and Practice*. John Wiley & Sons, Chichester, England, 1991.

[Goor92] A. J. van de Goor et T. J. W. Verhallen. Functional Testing of Current Microprocessors (applied to the Intel i860^{TM}). Dans *Proceedings IEEE International Test Conference (ITC)*, pages 684–695. 1992.

[Goor98] I. T. A.J. van de Goor. March Tests for Word-Oriented Memories. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*, page 501. 1998.

[Hamdioui98] S. Hamdioui et A. van de Goor. Consequences of Port Restrictions on Testing Two-Port Memories. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 63–72. 1998.

[Hatzimihail05] M. Hatzimihail, M. Psarakis, G. Xenoulis, D. Gizopoulos et A. Paschalis. Software-Based Self-Test for Pipelined Processors : A Case Study. Dans *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*. 2005.

[Hetherington99] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan et J. Rajski. Logic BIST for Large Industrial Desings : Real Issues and Case Studies. Dans *Proceedings IEEE International Test Conference (ITC)*. 1999.

[Huang01] J.-R. Huang, M. K. Iyer et K.-T. Cheng. A Self-Test Methodology for IP Cores in Bus-Based Programmable SOCs. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 198–203. Marina del Rey, CA, mai 2001.

[Hwang01] S. Hwang et J. Abraham. Reuse of Addressable System Bus for SOC Testing. Dans *Proceedings of the ASIC/SOC Conference*, pages 215–219. 2001.

[IEEE1500] IEEE1500. IEEE 1500 Web Site. http ://grouper.ieee.org/groups/1500/.

[Iyer02] M. K. Iyer et K.-T. Cheng. Software-Based Weighted Random Testing for IP Cores in Bus-Based Programmable SoCs. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*. April 2002.

[Jayaraman02] K. Jayaraman, V. M. Vedula et J. A. Abraham. Native Mode Functional Self-Test Generation for Systems-on-Chip. Dans *Proceedings of the 3rd International Symposium on Quality Electronic Design (ISQED)*, pages 280–285. 2002.

[Kapur03] R. Kapur. *CTL for Test Information of Digital ICs*. Kluwer Academic Publishers, 2003.

[Koeppe86] S. Koeppe. Modeling and Simulation of Delay Faults in CMOS Logic Circuits. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 530–536. 1986.

[Kranitis02a] N. Kranitis, D. Gizopoulos, A. Paschalis et Y. Zorian. Instruction-Based Self-Testing of Processor Cores. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*. 2002.

[Kranitis02b] N. Kranitis, A. Paschalis, D. Gizopoulos et Y. Zorian. Instruction-Based Self-Testing of Processor Cores. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. 2002.

[Kranitis03a] N. Kranitis, G.Xenoulis, D. Gizopoulos, A. Paschalis et Y. Zorian. Software-Based Self-Testing of Large Register Banks in RISC Processor Cores. Dans *LATW'2003 - IEEE Latin-American Test Workshop*. February 2003.

[Kranitis03b] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis et Y. Zorian. Low-Cost Software-Based Self-Testing of RISC Processor Cores. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. Munich, Germany, mars 2003.

[Kranitis03c] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos et Y. Zorian. Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 431–440. 2003.

[Kranitis05] N. Kranitis, A. Paschalis, D. Gizopoulos et G. Xenoulis. Software-Based Self-Testing of Embedded Processors. *IEEE Transactions on Computers*, 54(4) : pages 461–475, 2005.

[Lai01] W.-C. Lai, J.-R. Huang et K.-T. Cheng. Embedded-Software-Based Approach to Testing Crosstalk-Induced Faults at On Chip Buses. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 204–209. Marina del Rey, CA, mai 2001.

[Landrault96] C. Landrault. Test, testabilité et test intégré des circuits intégrés logiques. Rapport technique, LIRMM, novembre 1996.

[Lee05] K.-J. Lee, C.-Y. Chu et Y.-T. Hong. An embedded processor based SOC test platform. Dans *Proceedings International Symposium on Circuits and Systems (ISCAS)*, pages 2983–2986. 2005.

[Marinissen] E. J. Marinissen, V. Iyengar et K. Chakrabarty. ITC'02 SOC Test Benchmarks Web Site. http ://www.extra.research.philips.com/itc02socbenchm/.

[Marinissen98] E. Marinissen. A Structured And Scalable Mechanism for Test Access to Embedded Reusable Cores. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 284–293. Washington, DC, octobre 1998.

[McLaurin00] T. L. McLaurin et F. Frederick. The testability features of the MCF5407 containing the 4^{th} generation Coldfire microprocessor core. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 151–159. 2000.

[MIPS] MIPS. MIPS Technologies, Inc Web Site. http ://www.mips.com.

[Nadeau-Dostie90] B. Nadeau-Dostie, A. Silburt et V. K. Agarwal. Serial Interfacing for Embedded-Memory Testing. *IEEE Transactions on Computers*, 7(2) : pages 52–63, 1990.

[Nadeau-Dostie00] B. Nadeau-Dostie. *Design For At-Speed Test, Diagnosis and Measurement*. Kluwer Academic Publishers, 2000.

[Nahvi04] A. Nahvi, M. Ivanov. Indirect test architecture for SoC testing. *IEEE Transactions on Computer-Aided Design*, 23(7) : pages 1128–1142, July 2004.

[Navabi93] Z. Navabi. *VHDL : Analysis and modeling of digital systems*. McGraw-Hill, 1993.

[Nourani98] M. Nourani et C. Papachristou. Parallelism in Structural Fault Testing of Embedded Cores. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 15–20. Monterey, CA, avril 1998.

[NXP] NXP website. http ://www.nxp.com.

[Papachristou99] C. A. Papachristou, F. Martin et M. Nourani. Microprocessor based testing for core-based system on chip. Dans *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, pages 586–591. 1999.

[Parvathala02] P. Parvathala, K. Maneparambil et W. Lindsay. FRITS – A Microprocessor Functional BIST Method. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 590–598. Washington, DC, USA, 2002.

[Paschalis01] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis et Y. Zorian. Deterministic Software-Based Self-Testing of Embedded Processor Cores. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. Munich, Germany, mars 2001.

[Paschalis04] A. Paschalis et D. Gizopoulos. Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*. 2004.

[Patil92] S. Patil et J. Savir. Skewed-Load Transition Test : Part 2, Coverage. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 714–722. 1992.

[Press06] R. Press et J. Boyer. Easily Implement PLL Clock Switching for At-Speed Test. http ://www.chipdesignmag.com/display.php ?articleId=376&issueId=15, March 2006.

[Rabaey03] J. Rabaey, A. Chandrakasan et B. Nikolic. *Digital Integrated Circuits*. Prentice-Hall, 2003.

[Rajsuman99] R. Rajsuman. Testing a System-on-a-Chip with Embedded Microprocessor. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 499–508. Atlantic City, NJ, septembre 1999.

[Rhodehamel89] M. W. Rhodehamel. The Bus Interface and Paging Units of the i860(tm) Microprocessor. Dans *Proceedings International Conference on Computer Design (ICCD)*, pages 380–384. 1989.

[Savir92] J. Savir. Skewed-Load Transition Test : Part 1, Calculus. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 705–713. 1992.

[Savir94] J. Savir et S. Patil. On broad-side delay test. Dans *IEEE Transactions on VLSI Systems*, tome 2, page 368. IEEE Press, 1994.

[Shen98] J. Shen et J. A. Abraham. Native Mode Functional Test Generation For Processors with Applications to Self-Test and Design Validation. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 990–999. Washington, DC, octobre 1998.

[Smith85] G. L. Smith. Model for delay faults based upon paths. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 342–349. 1985.

[SoCLIB] SoCLIB. A modelisation and simulation plat-form for system on chip. http ://-soclib.lip6.fr.

[Stroud02] C. E. Stroud. *A Designer's Guide to Built-In Self-Test*. Kluwer Academic Publishers, 2002.

[SYNOPSYS] SYNOPSYS. Synopsys Web Site. http ://www.synopsys.com/.

[Tehranipour01] M. Tehranipour, Z. Navabi et S. Fakhraie. An efficient BIST method for testing of embedded SRAMs. Dans *Proceedings International Symposium on Circuits and Systems (ISCAS)*, tome 5, pages 73–76. May 2001.

[Tehranipour03] M. Tehranipour, Z. Navabi et S. Fakhraie. Systematic test program generation for SoC testing using embedded processor. Dans *Proceedings International Symposium on Circuits and Systems (ISCAS)*, tome 5, pages 541–544. May 2003.

[Tendolkar00] N. Tendolkar, R. Molyneaux, C. Pyron et R. Raina. At-speed testing of delay faults for Motorola's MPC7400, a PowerPC^{TM} microprocessor. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 3–8. 2000.

[Thatte80] S. Thatte et J. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers*, 29(6) : pages 429–441, june 1980.

[Tsai00] H. C. Tsai, K. T. Cheng et S. Bhawmik. On Improving Test Quality of Scan-Based BIST. *IEEE Transactions on Computer-Aided Design*, pages 928–938, 2000.

[Varma98] P. Varma et S. Bhatia. A Structured Test Re-Use Methodology for Core-Based System Chips. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 294–302. Washington, DC, octobre 1998.

[Vermeulen01] B. Vermeulen, S. Oostdijk et F. Bouwman. Test and Debug Strategy of the PNX8525 Nexperia^{TM} Digital Video Platform System Chip. Dans *Proceedings IEEE International Test Conference (ITC)*, pages 121–130. Baltimore, MD, octobre 2001.

[VSIA] VSIA. VSI Alliance Web Site. http ://www.vsi.org/.

[Wang04] S. Wang, S. T. Chakradhar et B. Kedarnath. Re-configurable embedded core test protocol. Dans *Proceedings IEEE Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 234–237. 2004.

[Wu99] C.-F. Wu, C.-T. Huang et C.-W. Wu. RAMSES : a fast memory fault simulator. Dans *Proc. Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 165–173. Albuquerque, novembre 1999.

[Zhao00] J. Zhao, S. Irrinki, M. Puri et F. Lombardi. Detection of Inter-Port Faults in Multi-Port Static RAMs. Dans *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 297–302. 2000.

[Zhou06] J. Zhou et H.-J. Wunderlich. Software-based self-test of processors under power constraints. Dans *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 430–435. 2006.

[Zorian02] Y. Zorian. What is Infrastructure IP ? *IEEE Design & Test of Computers*, 19(3) : pages 3–5, may-june 2002.

Liste des publications

[1] M. Benabdenbi, A. Greiner, F. Pêcheux, E. Viaud, and **M. Tuna**. STEPS : Experimenting a New Software-Based Strategy for Testing SoCs Containing P1500-Compliant IP Cores. In *Conference on Design, automation and test in Europe (***DATE'04***)*, pages 10712–10713, February 2004.

[2] **M. Tuna** and E. Viaud. STEPS : une approche logicielle pour le test des circuits integres sur puce (SoC). In *7th Journées Nationales du Réseau Doctoral de Microélectronique (***JNRDM'04***)*, pages 263–265, may 2004.

[3] M. Diaby, **M. Tuna**, J.L. Desbarbieux, and F. Wajsburt. High Level Synthesis Methodology from C to FPGA Used for a Network Protocol Communication. In *15th IEEE International Workshop on Rapid System Prototyping (***RSP'04***)*, pages 103–108, june 2004.

[4] **M. Tuna**, M. Benabdenbi, and A. Greiner. STESI : Testing wrapped IP cores using a dedicated Test Processor. In *3rd IEEE International Workshop on Infrastructure IP (***I-IP'05***)*, may 2005.

[5] **M. Tuna**, M. Benabdenbi, and A. Greiner. STESI : a new software-based strategy for testing socs containing wrapped ip cores. In *12th International Conference Mixed Design of Integrated Circuits and Systems (***MIXDES'05***)*, pages 459–464, june 2005.

[6] **M. Tuna** and M. Benabdenbi. Software-Based Self-Test of Register Files in RISC Processor Cores using March Algorithms. In *7th IEEE Latin America Test Workshop (***LATW'06***)*, pages 67–72, digest of papers, March 2006.

[7] **M. Tuna**, M. Benabdenbi, and A. Greiner. STESOC : A Software-Based Test-Access-Mechanism Controller. In *11th IEEE European Test Symposium (***ETS'06***)*, pages 91–96, digest of papers, May 2006.

[8] **M. Tuna**, M. Benabdenbi, and A. Greiner. T-Proc : An Embedded IEEE1500-Wrapped Cores Tester. In *2nd IEEE Conference on Ph.D. Research in Microelectronics and Electronics (***PRIME'06***)*, pages 493–496, June 2006.

[9] **M. Tuna**, O. Garcia, and M. Benabdenbi. Software-Based Self-Test Strategies for Memory Caches of RISC Processor Cores. In *8th IEEE Latin America Test Workshop (***LATW'07***)*, March 2007.

[10] **M. Tuna**, M. Benabdenbi, and A. Greiner. At-Speed Testing of Core-Based System-On-Chip Using an Embedded Micro-Tester. In *25th IEEE VLSI Test Symposium (***VTS'07***)*, page to be published, May 2007.