NATIONAL & KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS & TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Fault Detection Methodology for Caches in Reliable Modern VLSI Microprocessors based on Instruction Set Architectures

Georgios A. Theodorou

ATHENS
SEPTEMBER 2012

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μεθοδολογία Ανίχνευσης Ελαττωμάτων Κρυφών Μνημών για Αξιόπιστους Σύγχρονους VLSI Μικροεπεξεργαστές που βασίζεται σε Αρχιτεκτονικές Συνόλου Εντολών

Γεώργιος Α. Θεοδώρου

ΑΘΗΝΑ
ΣΕΠΤΕΜΒΡΙΟΣ 2012

PhD THESIS

Fault Detection Methodology for Caches in Reliable Modern VLSI Microprocessors based on Instruction Set Architectures

Georgios A. Theodorou

SUPERVISOR: Antonios Paschalis, Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:
Antonios Paschalis, Professor NKUA
Dimitrios Gizopoulos, Associate Professor NKUA
Konstantinos Halatsis, Emeritus Professor NKUA

SEVEN-MEMBER EXAMINATION COMMITTEE

Antonios Paschalis,
Professor NKUA

Dimitrios Gizopoulos,
Associate Professor NKUA

Konstantinos Halatsis,
Emeritus Professor NKUA

Aggeliki Arapoyanni,
Professor NKUA

Kiamal Pekmestzi,
Professor NTUA

Nectarios Koziris,
Associate Professor NTUA

Mihalis Psarakis,
Assistant Professor UNIPI

Examination Date 3/09/2012

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μεθοδολογία Ανίχνευσης Ελαττωμάτων Κρυφών Μνημών για Αξιόπιστους Σύγχρονους VLSI Μικροεπεξεργαστές που βασίζεται σε Αρχιτεκτονικές Συνόλου Εντολών

Γεώργιος Α. Θεοδώρου

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Αντώνιος Πασχάλης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Αντώνιος Πασχάλης, Καθηγητής ΕΚΠΑ
Δημήτριος Γκιζόπουλος, Αν. Καθηγητής ΕΚΠΑ
Κωνσταντίνος Χαλάτσης, Ομ. Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Αντώνιος Πασχάλης,
Καθηγητής ΕΚΠΑ

Δημήτριος Γκιζόπουλος,
Αν. Καθηγητής ΕΚΠΑ

Κωνσταντίνος Χαλάτσης,
Ομ. Καθηγητής ΕΚΠΑ

Αγγελική Αραπογιάννη,
Καθηγήτρια ΕΚΠΑ

Κιαμαλ Πεκμεστζή,
Καθηγητής ΕΜΠ

Νεκτάριος Κοζύρης,
Αν. Καθηγητής ΕΜΠ

Μιχάλης Ψαράκης,
Επ. Καθηγητής ΠΑΠΕΙ

Ημερομηνία εξέτασης 3/09/2012

ABSTRACT

The present PhD thesis introduces a low cost fault detection methodology for small embedded cache memories that is based on modern Instruction Set Architectures and is applied with Software-Based Self-Test (SBST) routines. The proposed methodology applies March tests through software to detect both storage faults when applied to caches that comprise Static Random Access Memories (SRAM) only, e.g. L1 caches, and comparison faults when applied to caches that apart from SRAM memories comprise Content Addressable Memories (CAM) too, e.g. Translation Lookaside Buffers (TLBs). The proposed methodology can be applied to all three cache associativity organizations: direct mapped, set-associative and full-associative and it does not depend on the cache write policy. The methodology leverages existing powerful mechanisms of modern ISAs by utilizing instructions that we call in this PhD thesis Direct Cache Access (DCA) instructions. Moreover, our methodology exploits the native performance monitoring hardware and the trap handling mechanisms which are available in modern microprocessors. By effectively combining these features of modern microprocessors the proposed methodology applies March write and read operations with lower cost (code size, execution time, system performance overhead) when compared with other proposed solutions in the literature for fault detection in caches through SBST. Moreover, the proposed Methodology applies March compare operations when needed (for CAM arrays) and verifies the test result with a compact response to comply with periodic on-line testing needs. Finally, a multithreaded optimization of the proposed methodology that targets multithreaded, multicore architectures is also presented in this thesis. The proposed multithreaded optimization exploits the thread level parallelism of multithreaded, multicore architectures and the low level multiple sub-bank organization of modern cache designs to speedup March tests while preserving the March test quality.

The proposed methodology was applied to three processor benchmarks: a) OpenRISC 1200 b) LEON3 and c) OpenSPARC T1. In detail, the methodology was applied to the L1 caches of all three processor benchmarks and to the TLBs of OpenSPARC T1 processor. The multithreaded optimization was demonstrated on the multithreaded, multicore OpenSPARC T1. Experimental results both for the test code size and test execution time of several March tests demonstrate the effectiveness of the proposed methodology, its high adaptability and the significant improvements in terms of test time (86% for instruction L1 cache, 87% for the data L1 cache, about 40% for D-TLB and about 82% for I-TLB) and test code size (83% for instruction L1 cache, 86% for the data L1 cache, 3% for D-TLB and 35% for I-TLB) when the methodology is applied to the same benchmarks (LEON3 for L1 caches and OpenSPARC T1 for TLBs) and such DCA instructions are exploited compared to SBST solutions that don’t utilize these types of instructions. Moreover, experimental results show a speedup of more than 1.7 (for two threads) and more than 3.7 (for four threads) in test time when the proposed multithreaded optimization is applied to the L1 caches of OpenSPARC T1.

Finally, a test evaluation framework was implemented for several on-line periodic test scenarios in order to evaluate the system performance overhead of the proposed methodology under typical workloads (PARSEC benchmark suite). Simulation results show a performance overhead of less than 11% in strict scenarios and less than 6% in regular scenarios (e.g. periodic testing of L1 caches of all cores every one minute) for multicore architectures.

SUBJECT AREA: Hardware and Computer Architecture

KEYWORDS: Microprocessor testing, Reliability, Software-based Self-test, On-line testing, Memory testing, March tests

ΠΕΡΙΛΗΨΗ

Η παρούσα διδακτορική διατριβή εισάγει μία χαμηλού κόστους μεθοδολογία για την ανίχνευση ελαττωμάτων σε μικρές ενσωματωμένες κρυφές μνήμες που βασίζεται σε σύγχρονες Αρχιτεκτονικές Συνόλου Εντολών (Instruction Set Architectures - ISAs) και εφαρμόζεται με λογισμικό αυτοδοκιμής. Η προτεινόμενη μεθοδολογία εφαρμόζει αλγορίθμους March μέσω λογισμικού για την ανίχνευση τόσο ελαττωμάτων αποθήκευσης όταν εφαρμόζεται σε κρυφές μνήμες που περιέχουν μόνο στατικές μνήμες τυχαίας προσπέλασης (Static Random Access Memories - SRAMs) όπως για παράδειγμα κρυφές μνήμες επιπέδου 1 (L1 caches), όσο και ελαττωμάτων σύγκρισης όταν εφαρμόζεται σε κρυφές μνήμες που περιέχουν εκτός από SRAM μνήμες και μνήμες διευθυνσιοδοτούμενες μέσω περιεχομένου (Content Addressable Memories - CAMs), όπως για παράδειγμα πλήρως συσχετιστικές κρυφές μνήμες αναζήτησης μετάφρασης (Translation Lookaside Buffers - TLBs). Η προτεινόμενη μεθοδολογία εφαρμόζεται και στις τρεις οργανώσεις συσχετιστικότητας κρυφής μνήμης: άμεσης απεικόνισης, συσχετιστική συνόλου και πλήρως συσχετιστική και είναι ανεξάρτητη της πολιτικής εγγραφής στο επόμενο επίπεδο της ιεραρχίας. Η μεθοδολογία αξιοποιεί υπάρχοντες ισχυρούς μηχανισμούς των μοντέρνων ISAs χρησιμοποιώντας ειδικές εντολές, που ονομάζονται στην παρούσα διατριβή Εντολές Άμεσης Προσπέλασης Κρυφής Μνήμης (Direct Cache Access Instructions - DCAs). Επιπλέον, η προτεινόμενη μεθοδολογία εκμεταλλεύεται τους έμφυτους μηχανισμούς καταγραφής απόδοσης (performance monitoring hardware) και τους μηχανισμούς χειρισμού παγίδων (trap handlers) που είναι διαθέσιμοι στους σύγχρονους επεξεργαστές. Συνδυάζοντας αποδοτικά αυτά τα χαρακτηριστικά των σύγχρονών επεξεργαστών, η προτεινόμενη μεθοδολογία εφαρμόζει τις λειτουργίες γραψίματος και διαβάσματος των αλγορίθμων March μέσω λογισμικού με χαμηλότερο κόστος (μέγεθος κώδικα, χρόνος εκτέλεσης, επιβάρυνση της απόδοσης του συστήματος) σε σχέση με άλλες προτεινόμενες λύσεις στην βιβλιογραφία για την ανίχνευση ελαττωμάτων κρυφών μνημών με τη χρήση λογισμικού αυτοδοκιμής. Επιπρόσθετα, η προτεινόμενη μεθοδολογία εφαρμόζει την λειτουργία σύγκρισης των αλγορίθμων March όταν αυτή απαιτείται (για μνήμες CAM) και επαληθεύει το αποτέλεσμα του ελέγχου μέσω σύντομης απόκρισης, ώστε να είναι συμβατή με τις απαιτήσεις του ελέγχου εντός λειτουργίας. Τέλος, στη διατριβή προτείνεται μία βελτιστοποίηση της μεθοδολογίας για πολυνηματικές, πολυπύρηνες αρχιτεκτονικές. Η προτεινόμενη βελτιστοποίηση αξιοποιεί την παραλληλία επιπέδου νήματος των πολυνηματικών, πολυπύρηνων αρχιτεκτονικών και την παράλληλη οργάνωση χαμηλού επιπέδου των σύγχρονων κρυφών μνημών σε πολλαπλές τράπεζες μνήμης για να επιταχύνει την εκτέλεση των αλγορίθμων March διατηρώντας παράλληλα την ποιότητα ανίχνευσης ελαττωμάτων.

Η προτεινόμενη μεθοδολογία εφαρμόστηκε σε τρεις επεξεργαστές: α) OpenRISC 1200 β) LEON3 και γ) OpenSPARC T1. Πιο συγκεκριμένα, η μεθοδολογία εφαρμόστηκε στις L1 caches και των τριών επεξεργαστών και στις TLBs του επεξεργαστή OpenSPARC T1. Η προτεινόμενη βελτιστοποίηση της παραλληλίας σε επίπεδο νήματος εφαρμόστηκε στον πολυνηματικό, πολυπύρηνο επεξεργαστή OpenSPARC T1. Πειραματικά αποτελέσματα τόσο για τον χρόνο εκτέλεσης, όσο και για το μέγεθος του κώδικα διαφόρων αλγορίθμων March αποδεικνύουν την αποτελεσματικότητα της προτεινόμενης μεθοδολογίας, την προσαρμοστικότητα της και τη σημαντική βελτίωση τόσο στο χρόνο εκτέλεσης (86% για την L1 cache εντολών, 87% για την L1 cache δεδομένων, περίπου 40% για την TLB δεδομένων και περίπου 82% για την TLB εντολών), όσο και στο μέγεθος του κώδικα (83% για την L1 cache εντολών, 86% για την L1 cache δεδομένων, 3% για την TLB δεδομένων και 35% για την TLB εντολών) όταν η μεθοδολογία εφαρμόζεται στους ίδιους επεξεργαστές αναφοράς (LEON3 για τις L1 caches και OpenSPARC T1 για τις TLBs) και αξιοποιούνται τέτοιες εντολές DCA σε σύγκριση με άλλες προτεινόμενες λύσεις στην βιβλιογραφία για την ανίχνευση ελαττωμάτων κρυφών μνημών που εφαρμόζονται με λογισμικό αυτοδοκιμής που δεν αξιοποιούν αυτές τις εντολές. Επιπρόσθετα, τα πειραματικά αποτελέσματα έδειξαν επιτάχυνση πάνω από 1.7 (για δύο νήματα) και 3.7 (για τέσσερα νήματα) του χρόνου εκτέλεσης, όταν η προτεινόμενη βελτιστοποίηση της παραλληλίας σε επίπεδο νήματος εφαρμόστηκε στις κρυφές μνήμες επιπέδου 1 του OpenSPARC T1.

Τέλος, υλοποιήθηκε ένα πλαίσιο αποτίμησης για διαφορετικά σενάρια περιοδικού ελέγχου εντός λειτουργίας του συστήματος (on-line) με το οποίο αξιολογήθηκε η επιβάρυνση της ανίχνευσης ελαττωμάτων στην απόδοση του συστήματος υπό την παρουσία τυπικού φορτίου εργασίας (σουίτα μετροπρογραμμάτων PARSEC). Οι προσομοιώσεις έδειξαν επιβάρυνση απόδοσης μικρότερη του 11% σε αυστηρά σενάρια και μικρότερη του 6% σε συνήθη σενάρια (π.χ. περιοδικός έλεγχος των L1 caches όλων των πυρήνων κάθε ένα λεπτό) για πολυπύρηνες αρχιτεκτονικές.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Υλικό και Αρχιτεκτονική Υπολογιστικών Συστημάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Έλεγχος μικροεπεξεργαστών, Αξιοπιστία, Ανίχνευση ελαττωμάτων που εφαρμόζεται με λογισμικό αυτοδοκιμής, Έλεγχος εντός λειτουργίας, Έλεγχος μνημών, Αλγόριθμοι March

Στους γονείς μου, την αδελφή μου και όλες τις φίλες και τους φίλους μου που με ενθάρρυναν και με στήριξαν για να ολοκληρώσω αυτό το δύσκολο έργο

To my parents, my sister and all my friends who encouraged and supported me to complete this demanding task

ACKNOWLEDGEMENTS / ΕΥΧΑΡΙΣΤΙΕΣ

Η εκπόνηση της διδακτορικής διατριβής αποτελεί μια μακρά, επίπονη αλλά εξαιρετικά δημιουργική και ευχάριστη διαδικασία, στην οποία πολλοί συνέβαλαν ποικιλοτρόπως, και τους οποίους θα ήθελα να ευχαριστήσω εκ βάθους καρδιάς.

Καταρχήν, τον Επιβλέποντα καθηγητή μου κ. Αντώνη Πασχάλη, ο οποίος υπήρξε πηγή έμπνευσης, ήδη από το προπτυχιακό επίπεδο σπουδών μέσω των μαθημάτων που διδάσκει, ώστε να ασχοληθώ με την περιοχή της αξιοπιστίας υπολογιστικών συστημάτων, της αρχιτεκτονικής επεξεργαστών και του υλικού υπολογιστών γενικότερα. Τον ευχαριστώ θερμά για την καθοδήγηση του, σε όλη την διάρκεια της εκπόνησης της διατριβής, την συνεχή και πολύπλευρη υποστήριξη του και την αμέριστη συμπαράσταση στις καλές αλλά κυρίως στις δύσκολες στιγμές που συνάντησα σε όλα αυτά τα χρόνια της προσπάθειας μου, την αμεσότητα καθώς και την διαθεσιμότητα του ακόμα σε περιόδους μεγάλου φόρτου εργασίας.

Θα ήθελα να ευχαριστήσω, επίσης τον Αναπληρωτή Καθηγητή κ. Δημήτριο Γκιζόπουλο για την ώθηση και τις πολύτιμες συμβουλές του και την συμπαράσταση του που βοήθησαν σημαντικά στην επιτυχή ολοκλήρωση της διατριβής. Ευχαριστώ ακόμα τον Ομότιμο Καθηγητή κ. Κωνσταντίνο Χαλάτση για την συμμετοχή του στην Τριμελή Επιτροπή Παρακολούθησης.

Επίσης, θα ήθελα να ευχαριστήσω θερμά το διδάκτορα Νεκτάριο Κρανίτη για τη συνεχή βοήθεια και συμβολή του στην εκπόνηση της διατριβής μου. Οι γνώσεις που μου μετέδωσε στο αρχικό στάδιο της διατριβής μου και οι καίριες επισημάνσεις, κατευθύνσεις και παραινέσεις του, όλα αυτά τα χρόνια, αποτελούν τον ακρογωνιαίο λίθο πάνω στον οποίο στηρίχτηκε η διατριβή μου.

Ακόμη ευχαριστώ τον πολύ καλό μου φίλο και διδάκτορα Αντρέα Μερεντίτη, με τον οποίο μοιραστήκαμε μια κοινή πορεία και συνεργαστήκαμε πολλές φορές από το επίπεδο των προπτυχιακών σπουδών μέχρι το διδακτορικό και την πολύ καλή μου φίλη διδάκτορα Ελένη Πατούνη. Τους ευχαριστώ και τους δύο θερμά γιατί υπήρξαν για χρόνια συνοδοιπόροι στην προσπάθεια μου. Μέσα από μια άριστη φιλία, από τον προπτυχιακό κύκλο σπουδών ακόμα, μοιραστήκαμε ερευνητικές και προσωπικές ανησυχίες για την επίλυση δυσκολιών και αντιξοοτήτων από κοινού όλα αυτά τα χρόνια.

Επιπρόσθετα, ευχαριστώ θερμά τους 14 εργοδότες (προϊστάμενοι τμημάτων ΤΕΙ, διευθυντές ΙΕΚ, διευθυντές ΚΕΚ, διευθυντές φροντιστηρίων κλπ.) για τους οποίους έχω εργαστεί όλα αυτά τα χρόνια παράλληλα με την εκπόνηση της διδακτορικής μου

διατριβής που με εμπιστεύτηκαν για να διδάξω στους σπουδαστές τους πάνω από 30 θεωρητικά και εργαστηριακά μαθήματα, εξασφαλίζοντας τον βιοπορισμό μου αλλά και αποκομίζοντας μοναδικές εμπειρίες στον χώρο της μεταλυκειακής εκπαίδευσης.

Βαθιά ευγνωμοσύνη νιώθω για την οικογένεια μου: α) τους γονείς μου Αργύρη και Αγγελική, οι αρχές των οποίων διαμόρφωσαν την προσωπικότητα μου και οι θυσίες τους μου έδωσαν την δυνατότητα να ολοκληρώσω τις προπτυχιακές και μεταπτυχιακές μου σπουδές απρόσκοπτα χωρίς να εργάζομαι αν και ερχόμενος από την επαρχία και σφυρηλάτησαν την αποφασιστικότητα μου για την επίτευξη ενός τόσο απαιτητικού στόχου όπως η ολοκλήρωση της διδακτορικής διατριβής, β) την αδελφή μου και συγκάτοικο μου Λένια για την αμέριστη συμπαράσταση της, όλα αυτά τα χρόνια.

Κλείνοντας, ευχαριστώ όλες τις φίλες και τους φίλους μου που σε μία τόσο μεγάλη διαδρομή στάθηκαν δίπλα μου, με στήριξαν και με ενθάρρυναν άμεσα ή έμμεσα με το δικό τους τρόπο διαμορφώνοντας ένα υγιές φιλικό περιβάλλον μέσα στο οποίο βίωσα χαρές, λύπες, συγκινήσεις, αντιθέσεις και πάνω από όλα αυθεντικές και μοναδικές προσωπικές στιγμές.

Πρωτίστως, ευχαριστώ τον Θεό που με βοήθησε να ξεπεράσω κάθε αντιξοότητα.

Γεώργιος Α. Θεοδώρου

Αθήνα, Σεπτέμβριος 2012

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

FOREWORD / ΠΡΟΛΟΓΟΣ

This thesis took place in Department of Informatics & Telecommunications in National & Kapodistrian University of Athens and was supervised by the professor Antonis Paschalis.

Η διδακτορική διατριβή εκπονήθηκε στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών υπό την επίβλεψη του Καθηγητή κ. Αντώνη Πασχάλη.

Η συγγραφή των κεφαλαίων της διδακτορικής διατριβής (κεφάλαια 1 έως 7) έγινε στην αγγλική γλώσσα ώστε να αποδοθεί με τη μεγαλύτερη δυνατή ακρίβεια η απαραίτητη τεχνική ορολογία. Κρίθηκε σημαντικό, τα ερευνητικά αποτελέσματα που παρουσιάζονται στην παρούσα διδακτορική διατριβή, τα οποία έχουν δημοσιευτεί σε διεθνή συνέδρια και περιοδικά στην αγγλική γλώσσα, να μην μεταφραστούν αλλά να μεταφερθούν αυτούσια στην αγγλική γλώσσα με στόχο, αφενός τη διατήρηση του περιεχομένου και της δομής και αφετέρου, την εξασφάλιση της αναγνωσιμότητας της διδακτορικής διατριβής από Έλληνες και ξένους επιστήμονες.

1. INTRODUCTION

1.1 Dependability and reliability in VLSI circuits

Dependability is defined as the ability to deliver service that can justifiably be trusted. This definition stresses the need for justification of trust. The alternate definition that provides the criterion for deciding if the service is dependable or not, is: the *dependability* of a system is its ability to avoid service failures that are more frequent and more severe than is acceptable [1].

It is usual to say that the dependability of a system should suffice for the dependence being placed on that system. The dependence of system A on system B, thus, represents the extent to which system A’s dependability is (or would be) affected by that of system B. The concept of dependence leads to that of trust, which can very conveniently be defined as accepted dependence.

As developed over the past three decades, dependability is an integrating concept that encompasses the following attributes:

- *availability*: readiness for correct service.
- *reliability*: continuity of correct service.
- *safety*: absence of catastrophic consequences on user(s) and the environment.
- *integrity*: absence of improper system alteration.
- *maintainability*: ability to undergo modifications and repairs.

Over the course of the past 50 years many means have been developed to attain the various attributes of dependability. Those means can be grouped into four major categories:

- *Fault prevention* means to prevent the occurrence or introduction of faults.
- *Fault tolerance* means to avoid service failures in the presence of faults.
- *Fault removal* means to reduce the number and severity of faults.
- *Fault forecasting* means to estimate the present number, the future incidence and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and the dependability and security specifications are adequate and that the system is likely to meet them.

Reliability is one of the attributes of dependability and is defined as the probability that a system will deliver its required services for a stated period of time. This definition of reliability highlights its statistical nature. Long term reliability of Very Large Scale Integrated (VLSI) circuits is becoming an important issue as the densities of VLSI circuits increase with shrink design rules. The assessment and improvement of reliability on the circuit level for VLSI circuits is a critical issue during the design flow and should be based on the understanding of the physical failure mechanisms observed in VLSI circuits, their modeling and the detection techniques of these failures.

Moreover, in case of VLSI circuits most of the schemes that aim to increase system's dependability by attaining any of the abovementioned four categories (fault prevention, fault removal, fault tolerance and fault forecasting) require fault detection techniques that can be implemented through VLSI testing approaches. Therefore, fault detection through VLSI testing techniques is an essential process for any scheme that aims to add dependability features to a VLSI circuit.

Reliability specialists often describe the failure probability distribution function of VLSI chips during lifetime using a graphical representation called the bathtub curve. The bathtub curve consists of three periods: an infant mortality period with a decreasing failure rate followed by a normal life period (also known as "grace period") with a low, relatively constant failure rate and concluding with a wear-out period that exhibits an increasing failure rate.

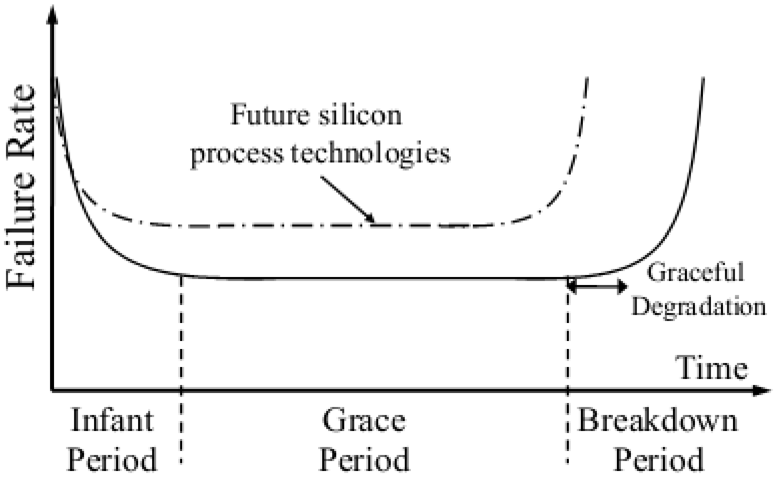


Figure 1: Bathtub curve

- *Infant mortality period*. The beginning of the product's lifetime is characterized by an initial high rate of device failures. These high failure rates are due to latent manufacturing defects that escape the manufacturing testing. These failures surface quickly when the manufacture-impaired devices are stressed as the products get into operation. However, the initial high failure rate declines rapidly as the remaining devices that pass the initial operating stress are more robust and less likely to fail.
- *Grace Period.* When early device failures are eliminated, the failure rate falls to a constant value where device failures occur sporadically due to the occasional break-down of weak transistors or interconnect that may be caused by either transient faults or single event effects. It is highly desirable that the grace period will dominate a product's lifetime since this is the period where the product exhibits the lowest failure rates and the highest reliability.
- *Breakdown Period*. After the grace period, device failures start to occur with increasing frequency over time due to age-related wearout. Many devices will enter this phase at roughly the same time, creating an avalanche effect and a quick rise in device failure rates. However, since not all devices will fail at once, it is likely that a short graceful degradation period exists over which a few initial device failures begin to signal the onset of the device breakdown period.

In deep submicron technology, the bathtub curve of the chips that are fabricated with these silicon process technologies is expected to shrink and exhibit higher failure rates. This will lead to products with shorter expected lifetimes. Furthermore, during their grace period, these products would be characterized by more frequent device failures caused by new operation faults.

1.2 Manufacturing testing: Basic principles

VLSI circuits are an integral part of any modern electronic system. Nowadays, such circuits contain millions or even billions of transistors, diodes and other components such as capacitors and resistors, together with inter-connections, within a very small area. The manufacturing process of such circuits is a complicated and time consuming process and the appearance of physical defects in VLSI circuits is inevitable. Such defects may be due to several deficiencies in the original silicon and in the manufacturing process. Examples of the former are impurities and dislocations, and examples of the latter are temperature fluctuations during wafer processing, open interconnections, open circuits, short circuits, and extra or missing transistors. The complexity of VLSI technology has reached the point where chips already contain over 1 billion transistors (on average 10 billion transistors per chip by 2015) on a single chip, with on-chip clock frequencies of over 3GHz according to 2011 ITRS Roadmap [2]. These trends have a profound effect on the cost and difficulty of chip testing. From the point of view of economics, it has been shown that the cost of detecting a faulty component is lower before the component is packaged and becomes part of a VLSI system. Therefore, testing is a very important aspect of any VLSI manufacturing process.

VLSI circuits can be classified into combinational circuits and sequential circuits. Therefore, a distinction can be made between testing each one of the two classes. Testing combinational circuits is much easier, since for each fault, one or two test vectors have to be applied; while the detection of faults in sequential circuits may require first to bring the circuit into a state in which the fault may be sensitized and observed.

Testing typically consists of applying a set of test stimuli to the inputs of the circuit under test (CUT) while analyzing the output responses, as illustrated in Figure 2. Circuits that produce the correct output responses for all input stimuli pass the test and are considered to be fault-free. Those circuits that fail to produce a correct response at any point during the test sequence are assumed to be faulty.

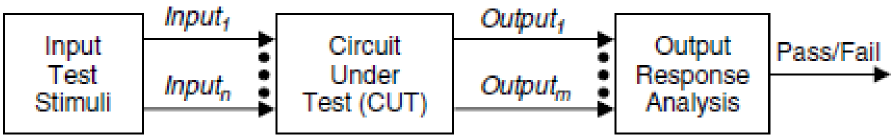


Figure 2: Basic testing approach

Testing is usually performed at various stages in the lifecycle of a VLSI device, including during the VLSI development process, the electronic system manufacturing process, and, in some cases, system-level operation.

There are two aspects of VLSI testing: fault detection and fault diagnosis. The testing process involves the application of test patterns to the circuit and comparing the response of the circuit with a precomputed expected response. If a chip is designed, fabricated, and tested, and fails the test, then there must be a certain cause for the failure [3]. Either (a) the test was wrong, (b) the fabrication process was faulty, (c) the design was incorrect, or (d) the specification had a problem; anything can go wrong.

The role of fault detection is to detect whether something went wrong, while the role of fault diagnosis is to determine exactly what went wrong and where the process needs to be altered.

VLSI testing can be classified into four types depending on the purpose it accomplishes [4]: characterization, production, burn-in, and incoming inspection.

- *Characterization*: also known as design debug or verification testing. This form of testing is performed on any new design before it is sent to production. It has to verify that the design is correct and meets the specifications; it also has to determine the exact limits of the device operating values. Functional tests are run during that phase, and comprehensive AC and DC parametric measurements are made. Tests are generally applied for the worst case, because they are easier to evaluate than average cases and devices passing these tests will work for any other conditions. Probing of the internal nodes of the chip may also be required during the design debug.
- *Production*: every fabricated chip is subjected to production tests, which must enforce the relevant quality requirements by determining whether the device meets the specifications; they are less comprehensive than characterization tests. The test may not cover all possible functions; however, they must have high fault coverage for the modeled faults. Fault diagnosis is not attempted and only a pass/fail decision is made.
- *Burn-in*: this ensures the reliability of devices, which pass production tests, by testing either continuously or periodically over a long period of time at elevated voltage and temperature [5]. Burn-in causes bad devices to actually fail. Two types of failures are isolated by burn-in: infant mortality and freak failures. Infant mortalities are screened out by a short term burn-in, typically 10-30 hours; they are often caused by a combination of sensitive design and process variation. Freak failures are devices having the same failure mechanisms as the reliable devices, but requiring long burn-in time, typically 100-1000 hours. During burn-in, production tests are applied at high temperatures and with an over-voltage power supply.
- *Incoming inspection*: incoming inspection is performed on purchased devices, before integrating them into a system. The most important purpose of incoming inspection tests is to avoid placing a defective device in a system, where the cost of diagnosis may far exceed the cost of incoming inspection.

Another classification for VLSI testing can be made depending on the type of targeted faults; either parametric or functional.

- *Parametric testing* is necessary to verify whether the chip meets DC and AC specifications. The DC parametric tests include maximum current, leakage, output drive current, thresholds levels, etc.; while the AC parametric tests include propagation delay, setup and hold times, functional speed, access time, and various rise and fall times.
- *Functional testing* determines whether the internal logic function of the chip behaves as intended; it checks for a proper operation of the design. Such test has to guarantee very high fault coverage of the modeled faults. Fault modeling of physical faults is a very important aspect in functional testing, since it turns the problem of test generation into a technology-independent problem. In addition, tests designed for modeled faults may be useful for detecting physical faults whose effect on circuit behavior is not well understood and/or too complex to be analyzed otherwise.

In the design hierarchy, a higher level description has fewer implementation details but more explicit functional information than a lower level description. The various levels of abstraction include behavioral (architecture), register-transfer, logical (gate), and physical (transistor) levels. The hierarchical design process lends itself to hierarchical test development, but the several fault models are more appropriate for particular levels of abstraction. Further down, we discuss test generation and the use of fault models at these various levels of abstraction.

- *Register-Transfer Level and Behavioral Level*. The methodology in common practice today is to design, simulate, and synthesize application-specific integrated circuits (ASICs) of millions of gates at the RTL. So-called “black boxes” or intellectual property (IP) cores are often incorporated in VLSI design, especially in SOC design, for which there may be very little, if any, structural information. Traditional automatic test pattern generation (ATPG) tools cannot effectively handle designs employing blocks for which the implementation detail is either unknown or subject to change; however, several approaches to test pattern generation at the RTL have been proposed. Most of these approaches are able to generate test patterns of good quality, sometimes comparable to gate-level ATPG tools. It is the lack of general applicability that prevents these approaches from being widely accepted. Although some experimental results have shown that RTL fault coverage can be quite close to fault coverage achieved at the gate level when designs are completed and mapped to a technology library, it is unrealistic to expect that stuck-at fault coverage at the RTL will be as high as at the gate level [6].

- *Gate level*. For decades, traditional IC test generation has been at the gate level based on the gate-level netlist. The stuck-at fault model can easily be applied for which many ATPG and fault simulation tools are commercially available. Very often the stuck- at fault model is also employed to evaluate the effectiveness of the input stimuli used for simulation-based design verification. As a result, the design verification stimuli are often also used for fault detection during manufacturing testing. In addition to the stuck-at fault model, delay fault models and delay testing have been traditionally based on the gate-level description. While bridging faults can be modeled at the gate level, practical selection of potential bridging fault sites requires physical design information. The gate-level description has advantages of functionality and tractability because it lies between the RTL and physical levels; however, it is now widely believed that test development at the gate level is not sufficient for deep submicron designs.
- *Switch level*. For standard cell-based VLSI implementations, transistor fault models (stuck-open and stuck-short) can be applied and evaluated based on the gate-level netlist. When the switch-level model for each gate in the netlist is substituted, we obtain an accurate abstraction of the netlist used for physical layout. In addition, transmission gate and tristate buffer faults can also be tested at the switch level. For example, it may be necessary to place buffers in parallel for improved drive capabilities. In most gate-level models, these buffers will appear as a single buffer, but it is possible to model a fault on any of the multiple buffers at the switch level. Furthermore, a defect-based test methodology can be more effective with a switch-level model of the circuit as it contains more detailed structural information than a gate-level abstraction and will yield a more accurate defect coverage analysis. Of course, the switch-level description is more complicated than the gate-level description for both ATPG and fault simulation.
- *Physical level*. The physical level of abstraction is the most important for VLSI testing because it provides the actual layout and routing information for the fabricated device and, hence, the most accurate information for delay faults, crosstalk effects, and bridging faults. For deep submicron IC chips, in order to characterize electrical properties of interconnections, a distributed *resistance-*

inductance-capacitance (RIC) model is based on the physical layout. This is then used to analyze and test for potential crosstalk problems. Furthermore, interconnect delays can be incorporated for more accurate delay fault analysis. One solution to the problem of determining likely bridging fault sites is to extract the capacitance between the wires from the physical design after layout and routing. This provides an accurate determination of those wires that are adjacent and, therefore, likely to sustain bridging faults.

1.3 On-Line testing: Basic principles

Various industrial sectors such as satellites, avionics, telecommunications, auto motives, medical electronics etc. have rapidly increasing needs for on-line testing during their system's lifetime. After manufacturing testing, VLSI chips are integrated in such systems and are placed in their natural environment, where operational faults may appear.

Such operational faults that may occur during lifetime in deep submicron technology are classified into the following three categories:

- *Permanent* operational faults that are infinitely active at the same location and reflect irreversible physical changes.
- *Intermittent* operational faults that appear repeatedly at the same location and cause errors in bursts only when they are active. These faults are induced by unstable or marginal hardware due to process variations and manufacturing residuals and are activated by environmental changes. In many cases, intermittent faults precede the occurrence of permanent faults.
- *Transient* operational faults appear irregularly at various locations and last short time. These faults are induced by neutron and alpha particles, power supply and interconnect noise, electromagnetic interference and electrostatic discharge.

On-line testing aims at detecting and/or correcting these operational faults by means of *concurrent* and *non-concurrent* test strategies [7].

- *Concurrent* on-line test strategies are used to detect all kinds of operational faults, while keeping the system in normal operation and are classified into the following four categories. *Hardware redundancy* strategies like duplication and comparison for fault detection and triple modular redundancy for error correction. *Information redundancy* strategies based on various coding schemes and self-checking design.

Both hardware and information-redundancy strategies have low fault-detection latency, but impose large to huge hardware overhead [8]. However, when a large increase in silicon area is not acceptable, the other two categories of concurrent on-line test strategies are used. *Time redundancy* strategies based either on recomputing using shifted/swapped operands or recomputing using duplication and comparison. *Software redundancy* strategies like N-version programming and software signature monitoring [9]. Both time and software redundancy strategies have higher fault-detection latency (compared to the first two strategies) and impose large to huge performance overhead. All concurrent on-line test strategies are high-cost solutions with hardware overhead, performance overhead but very low fault-detection latency.

- *Non-concurrent* test strategies such as on-line periodic testing are test strategies that trades off between fault-detection latency and performance overhead. In non-safety critical low-cost applications of embedded systems, there is no need for immediate detection of errors and, thus, no need for hardware, information, software, or time-redundancy mechanisms that increase significantly the system cost. In such embedded systems, detection of intermittent operational faults that cause errors in bursts only when they are active and may precede the occurrence of permanent faults, is much more important than detection of transient operational faults that appear once and last a short time. Therefore, on-line periodic testing is well suited to such embedded systems since it detects at low cost, not only permanent faults, but intermittent faults with very high probability. After fault detection, the system may reapply periodic testing several times to ensure that the fault is permanent or simply the system is restarted.

1.4 On-Line testing of embedded memories in microprocessor designs

During the past 30 years the semiconductor industry has been characterized by a steady path of constantly shrinking transistor geometries and increasing chip size. However, this technology achievement leads to new reliability challenges for modern systems that have not been considered in the past. Such reliability threats are either latent hardware defects that have not been detected by manufacturing tests or hardware defects that may occur during system operation by the increased soft error rate or by aging degradation effects. On-line testing schemes aim to detect such faults both in logic and memory modules of modern chips during their lifetime. Nowadays, in modern processors the relative chip area occupation of embedded cache memories is up to 90%. For example, UltraSPARC T1 [10] contains more than 170 large and small embedded cache memories. Thus, high quality cache memory on-line testing in modern processors is essential.

In manufacturing testing, Memory Build-in Self-Test (MBIST) is the industry standard for embedded memory testing. These MBIST schemes target the detection of memory functional faults [11] that are caused by cell spot defects. To achieve this, a large set of March tests is applied under different stress combinations to ensure detecting all possible faults in the caches of microprocessors. This set is not optimal and takes excessively long test time to be applied. For example in [12], a maximal test set of more than 50 March tests have been selected to kick off the process of constructing an optimal industrial cache test.

In on-line testing, parity and Error Correction Code (ECC) schemes are widely used to enhance embedded memory reliability and they are also utilized in embedded cache memories. These schemes are used to protect the memory arrays from soft errors. ECC schemes have two weaknesses i.e. fault accumulation effect and limited detect capability of memory functional faults [13]. Combined on-line MBIST schemes and ECC schemes can be used to overcome these weaknesses.

Programmable MBIST schemes [14], which have been integrated for manufacturing testing purposes, are reusable for on-line testing, but only a subset of the abovementioned March tests can be applied due to performance overhead limitations. The programmable MBIST schemes provide the flexibility to apply different March tests, as well as, future March tests for new memory fault models which is a critical feature for on-line testing. However, the reuse of MBIST schemes during on-line testing imposes much higher power density [15] that affects system's reliability.

Besides, small memory arrays that have size in the order of Kbytes (such as register files, FIFOs, small caches, cache tag arrays etc.) may not justify the cost of adding programmable MBIST schemes because of its impact on chip area and performance. Semiconductor industry has acknowledged this problem and industrial solutions have been proposed as a low-cost alternative to MBIST. For example, in [16], Macrotest, a scan-based technique was proposed to test a number of small embedded memories (including L1 caches) on the AMD Athlon™ processor during manufacturing testing. Moreover, in UltraSPARC T1 architecture a large number of small memory arrays lack an embedded MBIST scheme and are tested through Macrotest.

Level 1 (L1) cache Static Random Access Memory (SRAM) arrays and Translation Lookaside Buffers (TLBs) SRAM and Content Addressable Memory (CAM) arrays belong to this category of small memory arrays, since L1 cache sizes are up to 32Kbytes and TLBs are up to 128 entries in most of the modern processors. L1 cache arrays due to their size may totally or partially lack programmable MBIST circuitry (e.g. MBIST circuitry is not included in the tag array) whereas TLBs usually totally lack a MBIST scheme. Hardware defects in L1 cache arrays and TLB arrays during lifetime may cause either erroneous L1 cache and TLB misses that degrade the system's performance (defects in the tag array), or unpredicted system behaviour (defects in the data array). Hence, on-line testing for these arrays is essential to avoid system's performance degradation and erroneous behaviour.

1.5 Contribution of this thesis

Nowadays, on-line testing is essential for modern microprocessors to detect latent defects that either escape manufacturing testing or appear during system operation. Small memories, such as L1 caches and Translation Lookaside Buffers (TLBs) are not usually equipped with Memory Built-In Self-Test (MBIST) hardware. Software-Based Self-Test (SBST) is a flexible and low-cost solution for on-line March test application and error detection in such small memories. Although, L1 caches and TLBs are small components, their reliable operation is crucial for the system performance due to the large penalties caused when L1 cache or TLB misses occur.

The present PhD thesis introduces a low cost fault detection methodology for small embedded cache memories that is based on modern Instruction Set Architectures and is applied with SBST routines. The proposed methodology applies March tests through software to detect both storage faults [11] when applied to caches that comprise SRAM memories only, e.g. L1 caches, and comparison faults [17] when applied to caches that apart from SRAM memories comprise CAM memories too, e.g. TLBs. The proposed methodology can be applied to all three cache associativity organizations: direct mapped, set-associative and full-associative and it does not depend on the cache write policy. The methodology leverages existing powerful mechanisms of modern ISAs by utilizing instructions that we call in this PhD thesis Direct Cache Access (DCA) instructions. Moreover, our methodology exploits the native performance monitoring hardware and the trap handling mechanisms which are available in modern microprocessors. By effectively combining these features of modern microprocessors the proposed methodology applies March write and read operations with lower cost (code size, execution time, system performance overhead) when compared with other proposed solutions in the literature for fault detection in caches through SBST. Moreover, the proposed methodology applies March compare operations when needed (for CAM arrays) and verifies the test result with a compact response to comply with periodic on-line testing needs. Finally, a multithreaded optimization of the proposed methodology that targets multithreaded, multicore architectures is also presented in this thesis. The proposed multithreaded optimization exploits the thread level parallelism of multithreaded, multicore architectures and the low level multiple sub-bank organization of modern cache designs to speedup March tests while preserving the March test quality. Hence, in case of multithreaded, multicore architectures that can adopt the proposed multithreaded optimization, test time is further lowered and such SBST routines can be effectively executed periodically during the system’s lifetime with an acceptable performance overhead.

The proposed methodology was applied to three processor benchmarks: a) OpenRISC 1200 b) LEON3 and c) OpenSPARC T1. In detail, the methodology was applied to the L1 caches of all three processor benchmarks and to the TLBs of OpenSPARC T1 processor. The multithreaded optimization was demonstrated on the multithreaded, multicore OpenSPARC T1. Experimental results both for the test code size and test execution time of several March tests demonstrate the effectiveness of the proposed methodology, its high adaptability and the significant improvements in terms of test time and test code size when compared with other proposed solutions in the literature for fault detection in caches through SBST that do not exploit DCA instructions [75] [79].

Finally, a test evaluation framework was implemented in this thesis for several on-line periodic test scenarios in order to evaluate the system performance overhead of the proposed methodology. Simulation results indicate that the proposed March test implementation through SBST slightly influences the system’s performance, even in intensive test scenarios with high test frequency requirements.

The thesis is organized as follows:

Chapter 2 overviews the fundamental concepts of memory testing and March test algorithms both for single port SRAM and CAM memories. Firstly, the SRAM memory architecture is presented and the functional and electrical abstraction models that are mostly utilized in memory testing are described. Afterwards, the complete space of the memory faults for single-port memories is defined by exploiting the concept of fault primitives and functional fault models. Then, the faults in single port SRAMs are classified and the March tests that are suitable to detect these faults for single port SRAM faults are presented. Furthermore, the CAM memory architecture and the additional comparison faults that should be also considered in case of CAM memories are described along with the March tests that are suitable for detecting these additional comparison faults for CAM memories. Finally, a brief overview of the most common MBIST schemes that are utilized to apply these March tests is presented.

Chapter 3 briefly presents the features of Software-Based Self-Test approaches. Afterwards, the basic concept of how SBST is applied during manufacturing and on-line testing flow is presented. Additionally, the different application stages of SBST and the different requirements of each stage in terms of self-test code and data size, application time and power consumption are presented. Finally, several different SBST strategies that target processors and embedded memories proposed in the research literature are briefly discussed showing the evolution of SBST and experimental data sourcing from successful applications of the SBST approach are provided wherever available.

Chapter 4 introduces the proposed low cost fault detection methodology for small embedded cache memories that implements low cost March test operation for both L1 caches and TLBs by exploiting special debug-diagnostic DCA instructions. Firstly, it overviews all three cache organizations (direct mapped, set associative and fully associative), defines the cache arrays and presents the cache arrays testability challenges that occur when SBST approaches are utilized. Afterwards, it introduces the Direct Cache Access (DCA) instructions that are implemented in modern ISAs and are exploited to implement low cost March operations. The instruction fields, that an ideal DCA instruction should contain to gain direct access to all the cache arrays for applying any March operations either for SRAM or CAM memories, are defined and such ideal DCA instructions are composed both for L1 caches and TLBs. Moreover, existing DCA instructions that are close to ideal DCAs when combined are presented for RISC architectures, such as MIPS, ARM and SPARC architectures and for CISC architectures such as x86 architectures. Then, the main features of the proposed methodology are presented and a detailed description of the way that the proposed methodology implements every March operation is presented for data L1 cache, instruction L1 cache and TLBs respectively. Representative code snippets, that describe the way that March tests are implemented, are included for both data and instruction L1 caches and TLBs. Finally, the multithreaded optimization of our methodology is presented. The proposed optimization elaborates the low level multiple bank organization of modern cache designs to exploit the thread level parallelism of modern multithreaded, multicore processors and speedup March test execution time.

Chapter 5 presents the case study results that demonstrate the effectiveness of the proposed SBST program development methodology. The methodology has been applied to three processor benchmarks: a) OpenRISC 1200, b) LEON3 and c) OpenSPARC T1. A detailed description of the way that every March operation is implemented is presented for every processor benchmark. Moreover, statistical results for both test execution time and test code size are provided for every March test that has been implemented by utilizing the proposed methodology in all three benchmarks along with test coverage statistics for all cache arrays. Finally, the evaluation framework that was utilized to estimate the performance overhead of the proposed SBST routines is presented. Several on-line periodic testing scenarios are implemented and detailed statistics of the performance overhead introduced in a typical workload (PARSEC benchmark suite) under these test scenarios are presented.

Chapter 6 presents the comparison results of the proposed SBST methodology that exploits DCA instructions towards other SBST approaches in the literature that target small caches. Firstly, theoretical comparisons between the proposed SBST methodology and other SBST approaches are presented. These comparisons focus on the solutions that each SBST approach proposes to overcome the testability challenges of cache arrays. Afterwards, numerical statistical results are compared both for L1 caches and TLBs to demonstrate the effectiveness of the proposed methodology in terms of test execution time when applied to the same benchmarks with previous SBST approaches.

Finally, chapter 7 concludes the thesis and presents possible future research work to extend the proposed SBST methodology to other implicitly accessed memories such as L2 caches, SOC scratchpad memories etc. and to further optimize the implemented SBST routines under power constraints.

2. MEMORY TESTING – MARCH TESTS

2.1 SRAM memory architecture

A *single-port (SP)* SRAM memory is one which can only be read or written via a single circuit path at the time. In this thesis, only single port memories will be considered, since cache arrays are single port memory architectures. A *multi-port (MP) SRAM* memory is a memory that has multiple ports that are to be used to access memory cells simultaneously and independently of each other.

2.1.1 Memory models

A system may be described at a number of different levels of abstraction (Figure 3). Each level of abstraction is called a *model* of the system. Models help to simplify the explanation and treatment of systems by explicitly presenting information relevant only to the discussion about the system at that level, while hiding irrelevant information.

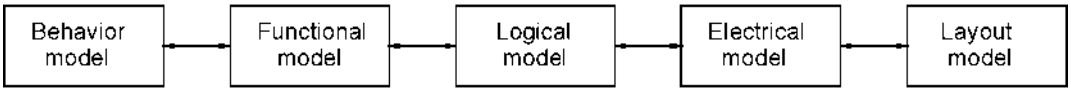


Figure 3: Memory models and levels of abstraction

The layout model is the one most closely related to the actual physical system; it assumes complete knowledge of the layout of the chip. As we move from right to left in Figure 3, the models become less representative of the physical world and more related to the way the system behaves, or in other words, less material and more abstract. A higher level of abstraction contains more explicit information about the way a system is expected to function and less about its buildup. It is possible to have a model that contains components from different levels of abstraction; this approach is called *mixed-level modeling.* With mixed-level modeling, one may focus on low- level details only in the area of interest in the system, while maintaining high-level models for the rest of the system.

The presented modeling levels in Figure 3 are explained as follows [18] [19]:

- *The behavioral model:* This is the highest modeling level and is based on the system specifications. At this level, the only information given is the relation between input and output signals while treating the system as a black box. A model at this level usually makes use of timing diagrams to convey information.
- *The functional model:* This model distinguishes functions that the system needs to fulfill in order to operate properly. At this level, the system is divided into several

interacting subsystems each with a specific function. Each subsystem is basically a black box called a *functional block* with its own behavioral model. The collective operation of the functional blocks results in the proper operation of the system as a whole.

- *The logical model:* This model is based on the logic gate representation of the system. At this level, simple Boolean relations are used to establish the desired system functionality. It is not the custom to model memories exclusively using logic gates, whereas logic gates are sometimes present in models of a higher or lower level of abstraction to serve special purposes.
- *The electrical model:* This model is based on the basic electrical components that build up the system. In semiconductor memories, the components are mostly transistors, resistors and capacitors. At this level, we are not only concerned with the logical interpretation of an electrical signal but also the actual electrical values of it.
- *The layout model:* This is the lowest modeling level available. It is directly related to the actual physical implementation of the system. At this level, all aspects of the system are taken into consideration, even the geometrical configuration, such as distances and thickness of lines, matters. For this reason, this model is also called the geometrical model. The data representing this model are rarely reported in the literature.

Paying a closer attention to the behavioral and the functional models reveals that there is a strong correspondence between the two. In fact, the behavioral model can be treated as a special case of the functional model, with the condition that only one function is presented, namely the function of the system itself. Therefore, some authors prefer to classify both modeling schemes as special cases of a more general model called the *structural model*. The structural model describes a system as a number of interconnected functional blocks. According to this definition, a behavioral model is a structural model with only one function, while a functional model is a structural model with more than one interconnected function.

2.1.2 Functional SP SRAM model

A typical SP memory consists of a memory cell array, two address decoders, read/write circuits, data flow and control circuits (Figure 4). The memory chip is connected to other devices through address lines, data lines, and control lines (i.e., read/write line, chip enable line, and power lines).

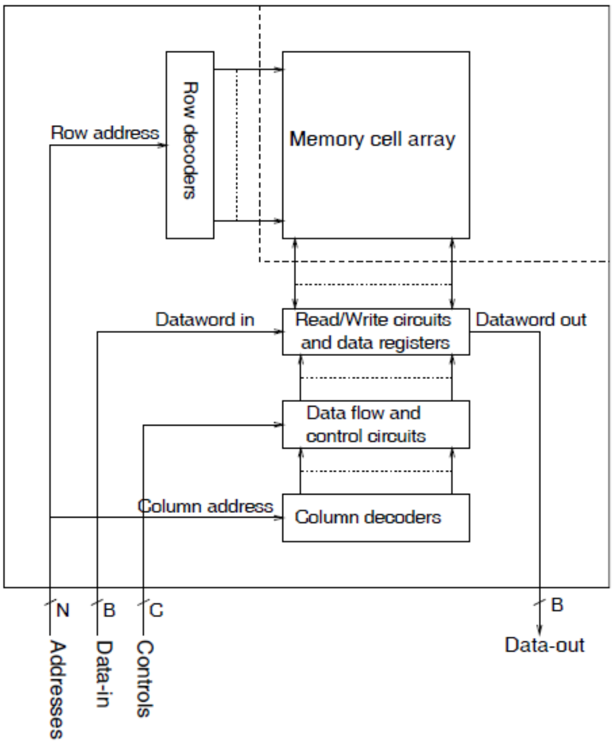


Figure 4: Functional model of a SP memory

The *memory cell array* is the most basic part of the memory. It consists of *n* cells, which are organized as an array of *R* rows and *C* columns. The number of rows can be any integer, but the number of columns is restricted: there is always an integer number of memory *words* in one row (i.e., *C mod B* = 0). Note that the memory cell array has a capacity of *R* x *C* bits.

The addresses are divided into high- and low-order bits. The high-order bits are connected to the *row decoder* which selects an appropriate row (*Word Line)* in the memory cell array, while the low-order bits are connected to the *column decoder* which selects the required columns *(Bit Lines).* The number of columns selected is *B,* which determines how many bits can be accessed during a read or a write operation.

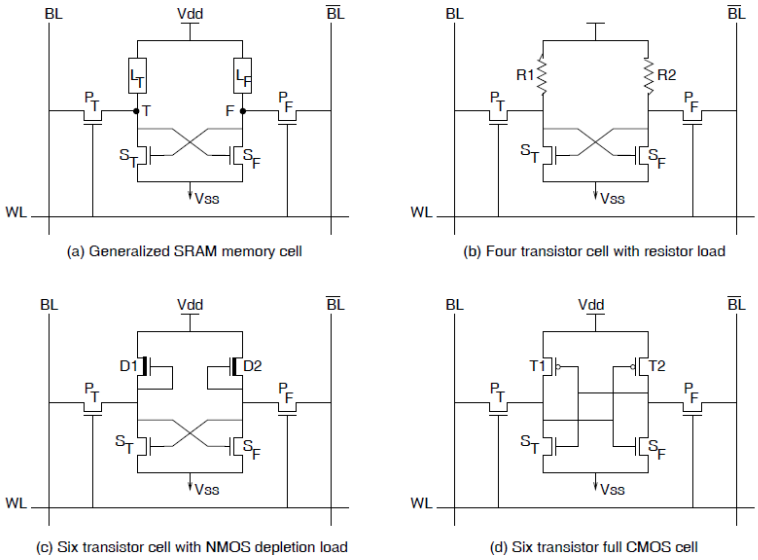
To read the desired memory cells, appropriate row and column select lines must be activated. The content of the selected cells are amplified by the read circuits, loaded into the data registers and presented on the data-out lines. Conversely, during a write operation, the data on the data-in lines is loaded into the data registers and written into the selected cells through the write circuits. Usually the data-in and data-out lines are combined to form bidirectional data lines, thus reducing the number of pins of the chip.

2.1.3 Electrical SP SRAM model

The blocks of the functional model for SP memories presented in Figure 4 will be opened such that the electrical properties will become visible. This will be done for memory cells, the address decoders, and the read/write circuits.

Memory cells

The memory cell is the most basic part of the memory. Its design depends on various factors, such as the memory application and the implementation style. For a SP SRAM, the memory cell is a *bistable* circuit, being driven into one of two states. After removing the driving stimulus, the circuit retains its state. An SRAM cell can have several configurations. Figure 5 shows the generalized SRAM cell, and three possible configurations.



(a) Generalized SRAM memory cell

(b) Four transistor cell with resistor load

(c) Six transistor cell with NMOS depletion load

(d) Six transistor full CMOS cell

Figure 5: Generalized SRAM cell, and various configurations of SRAM cells

As shown in Figure 5(a), the SRAM cell consists of two *load elements* (L_T and L_F), two *storage elements* (S_T and S_F*)*, and two *pass transistors* (P_T and P_F). Transistor S_T forms an inverter together with the load element L_T. This inverter is cross-coupled with the inverter formed by the transistor S_F and the load element L_F*;* therefore, forming a *latch.* This latch can be accessed for read and write operations, via the pass transistors P_T and P_F.

Data can be written into the cell by driving the lines BL and $\overline{BL}$ with data with complementary values, and thereafter driving the *word line (*WL*)* high. The cell will be forced to the state presented on BL and $\overline{BL}$, since the two lines are driven with more force than the force with which the cell retains its information. To read data from a cell, generally, first both lines BL and $\overline{BL}$ are *precharged* to a high level, after which the desired WL is driven high. At that time the data in the cell will pull one of the two bit lines low. This difference signal on the BL and $\overline{BL}$ lines is amplified by the read circuit, and read out through the data register. It should be noted that reading an SRAM cell is a non-destructive process; i.e., after the read operation the logic state of the cell remains the same.

The load devices may consist of polysilicon resistors, either enhancement or depletion mode transistors, or PMOS transistors. Figure 5 (b) shows the SRAM cell with polysilicon load devices. This cell requires less silicon area than the two other configurations. However, it has a higher current when it is not being accessed, since a small amount of current always flows through the resistor. When the load element is a PMOS transistor (Figure 5 (d)), then the resulting CMOS cell has essentially no current drain through the cell, except when it is switching because either the NMOS or the PMOS transistor is always off. The disadvantage of the CMOS cell is that it requires more processing steps because of the presence of NMOS and PMOS transistors. Figure 5 (c) shows a six-device SRAM cell using depletion mode load transistors. It should be noted that the depletion mode transistor can also be replaced with an enhancement mode transistor, but the depletion load is normally used since it has better switching performance, higher impedance, and is relatively insensitive to power supply variations.

Address decoders

Address decoders are used to access particular cells in the memory cell array. In order to reduce the size of the decoders and the length of the word and bit lines, two dimensional addressing schemes are used within the chip, demanding a row decoder for the word lines and a column decoder for the bit lines.

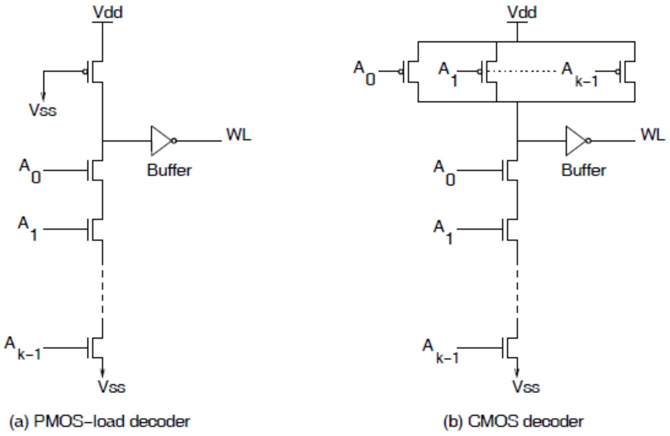


Figure 6: Static row decoders

A row decoder is needed to select one row out of the set of rows in the memory array. Figure 6 shows two basic *static row decoders*, namely a *PMOS-load decoder* [20] and a *CMOS decoder.* The inputs of the decoders are the address bits A_0 through A_{k-1} or their complements, while the output is the word line. When the row decoder selects a word line, all cells along that word line are active and put their data on the bit lines. Note that the address lines are connected only to the NMOS transistors in a PMOS-load decoder; while they are connected to both PMOS and NMOS transistors in a CMOS decoder. Therefore, the address load capacitance caused by the gates in a PMOS-load decoder is almost half of that in a CMOS decoder. This implies that a smaller delay time can be obtained by using a PMOS-load decoder. The CMOS decoder has the advantage of drawing no static current, but as it requires an equal number of PMOS and NMOS transistors, it occupies a larger area.

Dynamic or *clocked decoders* are also used to decode word lines. Figure 7 shows two dynamic row decoders. Generally, such decoders combine compact layout with zero static current consumption; power is dissipated in the selected decoder only during the brief period of an address transition. The decoder of Figure 7 operates as follows: in the precharge phase, the transistor *Q*1 is turned on to precharge the common line connected to the address decoding transistors. If all the address bits, A_0 through A_{k-1} are zeros, transistor *Q*6 drives the WL line to '1'. The signal EN enables the transmission gate such that the decoder selects the word line only after all address lines are stable. A column decoder selects *B* bit line(s) (or bit line pairs) out of the set of bit lines (or bit line pairs) of the selected row.

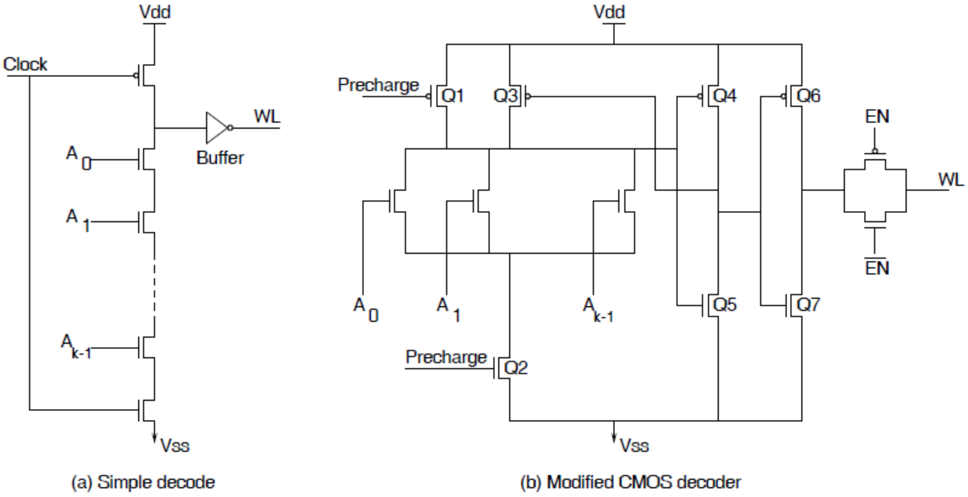


Figure 7: Dynamic row decoders

Depending on the memory application, different column decoders are designed. Figure 8(a) shows a *tree decoder*, which is desirable for *single ended* memories; i.e., memories which use only a single bit line for read and write operations. This circuit has the advantage of being simple, however it operates slowly. Figure 8(b) shows another column decoder, which is based on the PMOS-load decoder of Figure 2.7. The output of the decoder goes to an inverter, where the output signal is amplified, after which it moves on to the column switch MOS transistors. This circuit has the advantage of being compact.

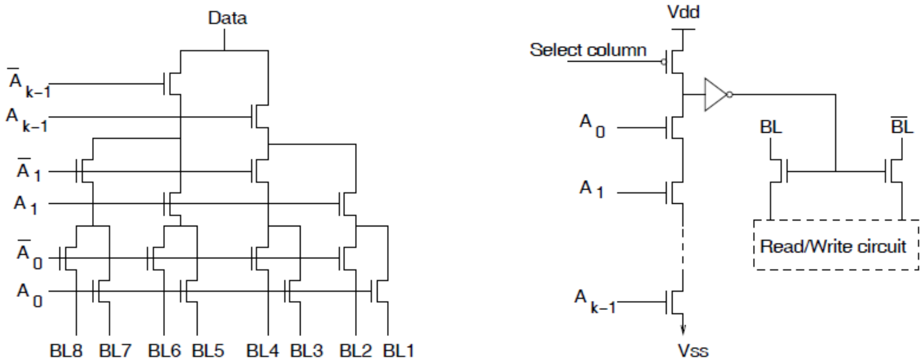


Figure 8: Column decoders

Read/Write circuitry

Once a particular single or pair of bit lines have been selected, depending on the SP memory cell structure, a circuitry is required to write or to read the cells. Typical write circuits are shown in Figure 9. Circuit (a) consists of a pair of inverters and a pass gate with a write control input signal; while circuit (*b*) consists of a pair of NAND gates. The data to be written on 'data in' line is presented on BL and $\overline{BL}$.

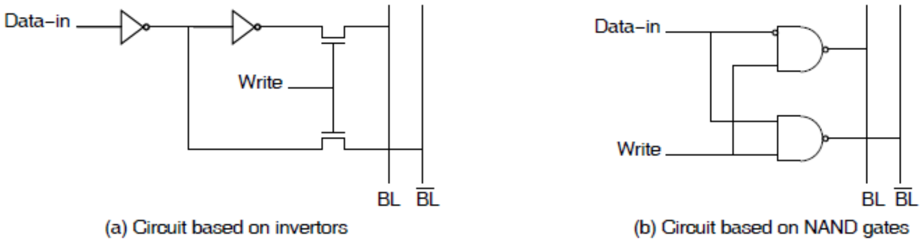

Data–in
Write
BL $\overline{BL}$
(a) Circuit based on invertors
Data–in
Write
BL $\overline{BL}$
(b) Circuit based on NAND gates

Figure 9: Memory write circuitries

The read circuitry is more complex than the write circuitry and depends on the type of the memory cells to be read, namely *single ended* or *differential.* In addition, it can be based on a *voltage mode* or a *current mode* signal transporting technique. Figure 10 shows two voltage mode sense amplifiers, namely a single ended PMOS differential sense amplifier, and a double-ended PMOS cross-coupled amplifier. In circuit (a), when the data on BL is ’1’, the transistor M1 turns on, and the transistor Q*2* drives the Out line to ’1’ while when the data on BL is ’0’, transistor M2 turns on, and drives the Out line to ’0’. In circuit (b), the voltages of the Out and $\overline{\text{Out}}$ lines control the gates of the PMOS transistors *Q*1 and *Q*2 such that the output voltage transitions are accelerated.

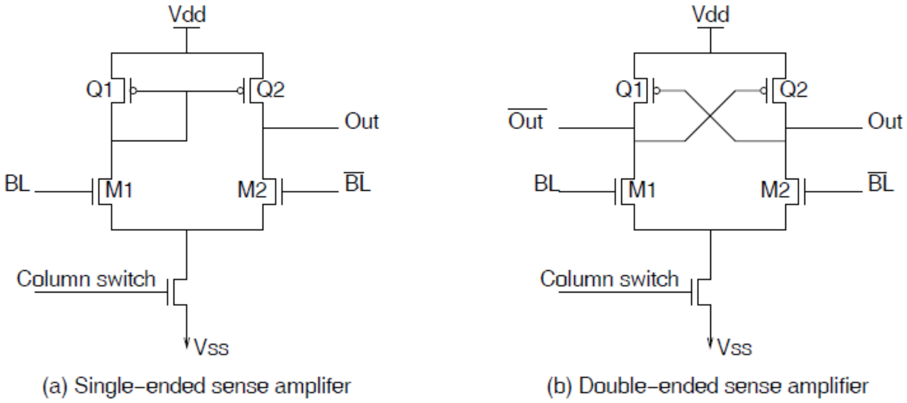


Figure 10: Voltage mode sense amplifiers

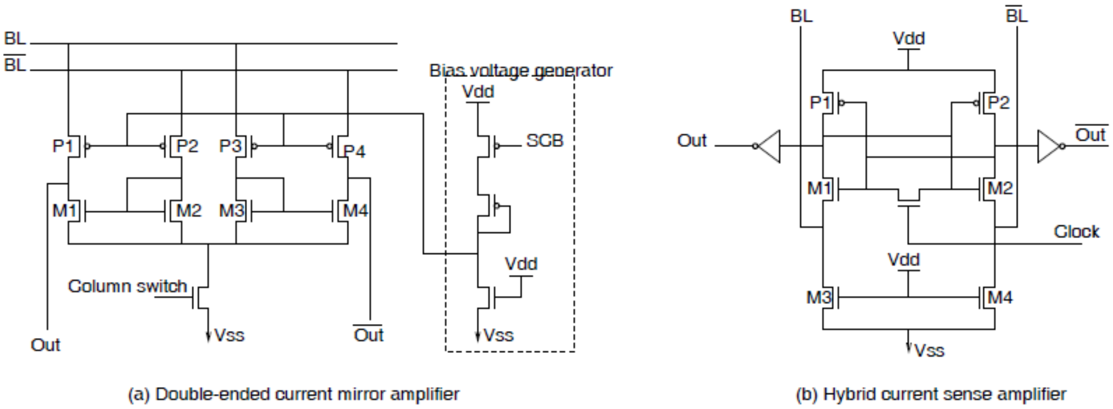


Figure 11: Current mode sense amplifiers

Current mode sense amplifiers operate generally faster than voltage mode ones. Figure 11 shows two current mode sense amplifies: a *double-ended current-mirror* amplifier and a *hybrid current sense* amplifier.

The double-ended current-mirror amplifier uses a bias voltage generator to provide an appropriate voltage to the PMOS transistors P1 though P4 so that they operate near the saturation region (Figure 11(a)). When no cell is selected, the current (say I_0) that flows from each bit line to the amplifier is the same. Therefore the current that flows through transistors P1-P4 in the amplifier is also the same ($I_0/2$). When a cell is selected, a small amount of current ΔI flows from one bit line to the cell. For example, ΔI flows from BL to the cell, the current flowing from BL to the amplifier will be reduced to I_0 - ΔI*,* while the current flowing from BL will remain to I_0*.* Consequently, the current flowing through PI and P3 will be reduced to (I_0 - ΔI)/2, while that flowing through P2 and P4 remains at $I_0/2$*.* The current which flows through M1 is $I_0/2$ because the current flowing through P2 and M2 is the same, and M1 is the current mirror of M2. Therefore the load capacitance of output line Out will be discharged by M1, which draws more current than P1 provides. In almost the same manner, the load capacitance of Out will be charged up by PI. In this way, the voltage of $\overline{\text{Out}}$ drops while the voltage of Out rises, and the voltage swing is obtained between the two data output lines.

The hybrid current sense amplifier (Figure 11 (b)) operates in two phases: equalization and sensing phase. During the equalization phase, the clock signal is high to precharge the output nodes to equal potentials. During the sensing phase, a particular memory cell is selected and the cell's node with low level draws a current from the corresponding bit line. The differential current signal then appears at BL and Out*.* Once the clock signal is low, the differential current flows through M1 and M2, and charging the small equivalent capacitances at the drains of M1 and M2. A small differential voltage will appear across the two drains and be amplified to a CMOS level voltage by the positive feedback effect of the cross coupled circuit.

2.2 Space of memory faults for SRAMs

2.2.1 Concept of fault primitive

Intuitively, a functional fault model is defined as a description of the failure of the memory to fulfill its functional specifications. This definition of a fault model is not a precise one since it does not indicate which functional specifications should be taken into account. Still, the definition specifies the intuitive meaning of a fault model and the way it should be viewed. The term ’functional specifications’ should be understood in a rather general sense. It should be detailed enough to describe the contents of individual memory cells.

By performing a number of memory operations and observing the behavior of any component functionally modeled in the memory, functional faults can be defined as the deviation of the observed behavior from the specified one under the performed operation(s). Therefore, the two basic ingredients to any fault model are:

1. A list of performed memory operations.
2. A list of corresponding deviations in the observed behavior from the expected one.

Any list of performed operations on the memory is called an *operation sequence.* An operation sequence that results in a difference between the observed and the expected memory behavior is called a *sensitizing operation sequence (SOS).* The observed memory behavior that deviates from the expected one is called a *faulty behavior.* When inspecting the memory for possible faulty behavior, not all the functional specifications are taken into account and compared with the actual memory behavior. Rather, a very limited subset of functional parameters is selected as most relevant to describe the faulty behavior of the memory. Throughout the 1980s and during the first half of the 1990s, the only functional parameter considered relevant to the faulty behavior was the stored logic value in the cell. Recently, another functional parameter, the output value of a read operation, was also considered to be relevant to describe the faulty behavior.

Thus in order to specify a certain fault, one has to specify the SOS, together with the corresponding faulty behavior. This combination for a single fault behavior is called a *Fault Primitive (FP)* [21], and is denoted as < *S/F/R >. S* describes the SOS that sensitizes the fault, *F* describes the value or the behavior of the faulty cell (e.g., the cell flips from 0 to 1), while *R* describes the logic output level of a read operation (e.g., 0).

The concept of a FP allows for establishing a complete framework of all memory faults, since for all allowed operation sequences in the memory, one can derive all possible faulty behaviors. In addition, the concept of a FP makes it possible to give a precise definition of a *functional fault model (FFM)* as it has to be understood for memory devices [21]:

A functional fault model is a non-empty set of fault primitives

This definition of a FFM still depends on the selected functional parameters to be observed in the FPs. Yet, this dependence is now precisely known once the FPs are defined. Since a fault model is defined as a set of FPs, it is expected that FFMs would inherit the properties of FPs. For example, if a FFM is defined as a collection of single cell FPs, then the FFM is a single cell fault. If a FFM is defined as a collection of 2-operation (i.e., the SOS consist of two sequential operations) FPs, then the FFM is also called a 2-operation fault.

The situation becomes more complicated if a FFM consists of FPs classified into inconsistent classes (e.g., single cell and two-cell FPs). In this case, the FFM is not described by a single term but by the classes of its constituent FPs. Therefore, a FFM that consists of single cell and two-cell FPs, for example, is described as a single and two-cell FFM.

2.2.2 Classification of fault primitives

Figure 12 shows the different classifications of the FPs. They can be classified based on:

1. The ways the FPs manifest themselves, into *simple* and *linked* faults.
2. The number of *sequential* operations required in the SOS, into *static* and *dynamic* faults.
3. The number of *simultaneous* operations required in the SOS, into *single-port* and *multi-port* faults.
4. The numbers of different cells the FPs do involve, into *single-cell* and *multi-cell* faults.

It is important to note that the four ways of classifying fault primitives are independent since their definition is based on independent factors of the SOS (Figure 12). As a result, a dynamic fault primitive can be single-port or multi-port, single-cell or multi-cell. The same is true for linked faults; they can be static or dynamic, and each of them can be single-port or multi-port, single-cell or multi-cell.

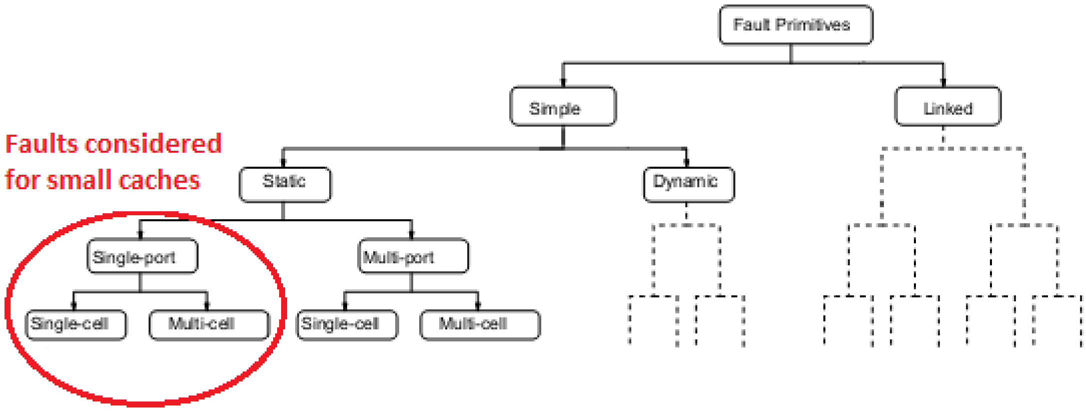


Figure 12 : Summary of fault primitive classification

2.2.3 Single versus linked faults

Depending on the way FPs manifests, they can be divided into *simple faults* and *linked faults.*

- *Simple faults:* These are faults which cannot influence the behavior of each other. That means that the behavior of a simple fault cannot change the behavior of another one; therefore *masking* cannot occur.
- *Linked faults:* These are faults that do influence the behavior of each other. That means that the behavior of a certain fault can change the behavior of another one such that *masking* can occur [22]. Note that linked faults consist of two or more simple faults. In order to get more insight into linked faults, the following example will be given. Assume that the application of an operation to a cell c_1 will cause a fault in a cell c_v (i.e., the cell flips); and that the application of an operation to a cell c_2 will cause a fault in the same cell c_v, but with a fault effect opposite to that caused by cell c_1. If now first an operation is applied to cell c_1, and thereafter to cell c_2, then the net result is that the fault effect of cell c_1 is masked by the fault effect of cell c_2; i.e., no fault effect is then visible in cell c_v.

2.2.4 Static versus dynamic faults

Let #O be defined as the number of different operations performed *sequentially* in a SOS. For example, if a single read operation applied to a certain cell causes the same cell to flip, then #O = 1. Depending on #O, FPs can be divided into *static* and *dynamic* faults:

- *Static faults:* These are FPs sensitized by performing *at the most* one operation; that is #O <= 1. For example, the state of the cell is always stuck at *one* (#O = 0), a read operation to a certain cell causes the same cell to flip (#O = 1), etc.
- *Dynamic faults:* These are FPs that can only be sensitized by performing more than one operation sequentially; that is #O > 1. Depending on #O, a further classification can be made between *2-operation dynamic FPs* whereby #O = 2, *3-operation dynamic FPs* whereby #*O* = 3, etc.

2.2.5 Single-port versus multi-port faults

Let #P be defined as the number of ports required *simultaneously* to apply a SOS. For example, if a single read operation applied to cell c_1 causes the same cell to flip, then #*P* = 1; if two *simultaneous* read operations applied to the cell cause the same cell to flip, then #*P* = 2. Depending on #*P,* FPs can be divided into *single-port* faults, and *multi-port* faults.

- *Single-port faults (1PFs):* These are FPs that require *at the most* one port in order to be sensitized; that is #P < 1. Note that single-port faults can be sensitized in single-port memories as well as in multi-port memories.
- *Multi-port faults (pPFs):* These are FPs that can be only sensitized by performing two or more simultaneous operations via the different ports. Depending on #P, the multi-port faults can be further divided into:
    - *Two-port faults (2PFs):* These are FPs that can be only sensitized by performing two simultaneous operations via two different ports; that is #*P* = 2. Note that 2PFs can be sensitized in any multi-port memory with *p > 2 (p* denotes the number of ports).
    - *Three-port faults (3PFs):* These are FPs that can only be sensitized by performing three simultaneous operations via three different ports; that is #P = 3. Note that 3PFs can be sensitized in any multi-port memory with *p* > 3,
    - etc.

In this thesis we will consider only single port faults, since cache arrays are single port memories.

2.2.6 Single-cell versus multi-cell faults

Let #C be defined as the number of different cells accessed during a SOS. For example, if the operation sequence consists only of a single read operation applied to a single cell, then #C = 1; if the operation sequence consists of two single read operations applied sequentially to two different cells, then #C = 2; etc. Depending on *#C,* FPs can be divided into *single cell faults* and *multi-cell faults* (i.e., *coupling faults*).

- *Single-cell faults*: These are FPs involving only a single cell. They have the property that the cell used for sensitizing the fault (by applying the SOS) is the *same* as where the fault appears.
- *Coupling faults*: These are FPs that involve more than one cell; they have the property that the cell(s) which sensitizes (or contribute for sensitizing) the fault (e.g., by applying the SOS) is *different* from the cell where the fault appears. Depending on *#C,* this class can be further divided into *two-coupling fault primitives* whereby *#C* = 2, *3-coupling fault primitives* whereby *#C* = 3, etc.

2.3 Single-port faults for SRAMs

Single-port faults occur in single-port memories such as cache memories and can be divided into single-cell FPs and multi-cell FPs. Single-cell FPs are FPs involving a single cell; while multi-cell FPs are FPs involving more than one cell. For multi-cell FPs, we will restrict our analysis to two-cell FPs (i.e., two-coupling FPs), because they are considered an important class in single-port SRAM faults.

Figure 13 shows the two classes considered for 1PFs. *Single-port faults involving a single cell (1PF1s)* have the property that the cell used for sensitizing the fault is the same cell as where the fault appears. On the other hand, *single-port faults involving two cells (1PF2)* are divided into three types, depending on the cell to which the sensitizing operation is applied.

- The 1PF2s: It has the property that the *state* of the *aggressor cell (a-cell c_a),* rather than an operation applied to c_a, sensitizes a fault in the *victim cell (v-cell c_v).* Note that no operation is required in that case; the subscript ‘s’ in the notation 1PF2s stands for ‘state’.
- The 1PF2a: It has the property that the application of a single-port operation (solid arrow in the figure) to the *a-cell* sensitizes a fault in the v-cell.

- The 1PF2v: It has the property that the application of a single-port operation to the *v-cell*, with the a-cell in a certain state, sensitizes a fault in the v-cell.

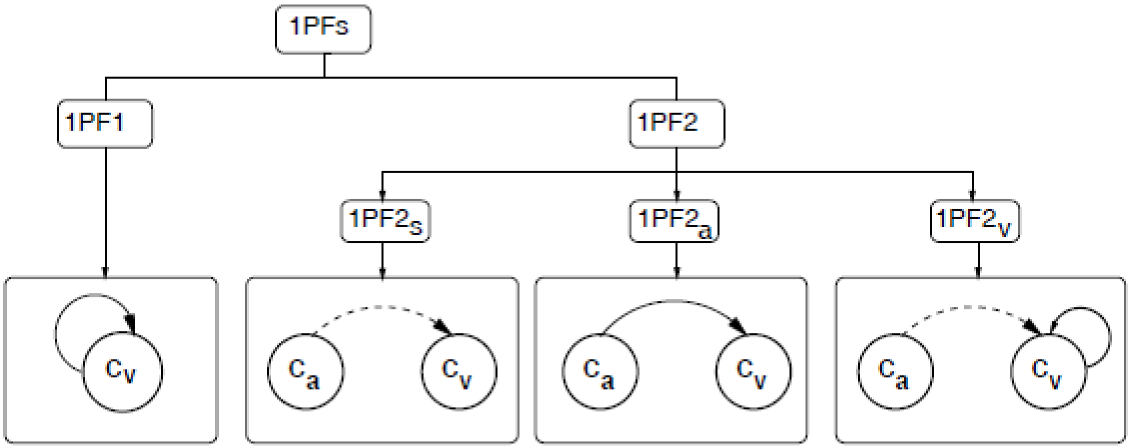


Figure 13: Classification of 1PFs

In the rest of this section, the domain of all possible 1PFs and 2PFs will be presented. For each class, first the complete list of FPs is given; thereafter the list will be compiled into FFMs.

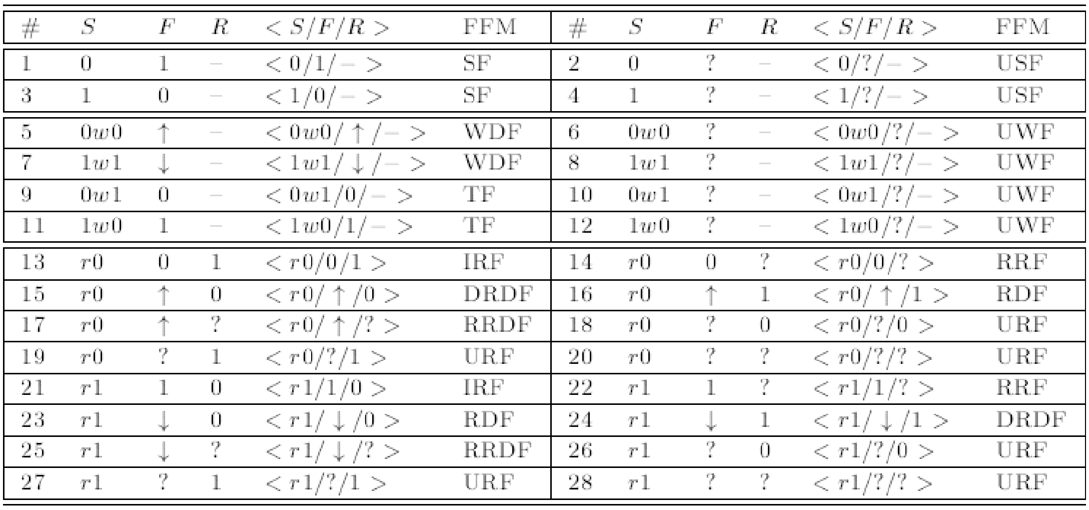
2.3.1 Single-cell fault primitives

Before listing the possible single-cell FPs (1PF1), a precise compact notation, which will prevent ambiguities and misunderstandings, will be introduced.

- *<S/F/R>* (or *<S/F/R>*$_V$): denotes an FP involving a single-cell (i.e. 1PF1s); the cell c_v (victim cell) used to sensitize a fault is the same cell as where the fault appears.
- *S* describes the value/operation *sensitizing* the fault; *S* ∊ {0, 1, 0w0, 1w1, 0w1, 1w0, r0, r1}, whereby 0(1) denotes a *zero (one)* value, 0w0 (1w1) denotes a write 0(1) operation to a cell which contains a 0 (1), 0w1 (1w0) denotes an up (down) transition write operation, and r0 (r1) denotes a read 0(1) operation. If the fault effect of *S* appears after a time T, then the sensitizing operation is given as S_T.
- *F* describes the value of the *faulty* cell (v-cell); *F* ∊ {0, 1, ↑, ↓, ?} whereby ↑(↓) denotes an up (down) transition due to a certain sensitizing *operation*; and '?' denotes an *undefined* state of the cell (e.g., the voltage of the true and the false node of the cell are almost the same).
- *R* describes the logical value which appears at the output of the SRAM if the sensitizing operation applied to the v-cell is a *read* operation: *R* ∊ {0, 1, ?, - } whereby '?' denotes a *random* logic value. A random logic value can occur if the voltage

difference between the bit lines (used by the sense amplifier) is very small. A ‘-’ in *R* means that the output data is not applicable; e.g., if *S = w0,* then no data will appear at the memory output, and for that reason *R* is replaced by a ‘-’. It is interesting to note here that the word *undefined* is used for the state of the cell (for *F* =?), and *random* is used for the read data value (for *R* =?); these words will be used later to give names for the introduced FFMs.

Table 1: Complete set of 1FP1s



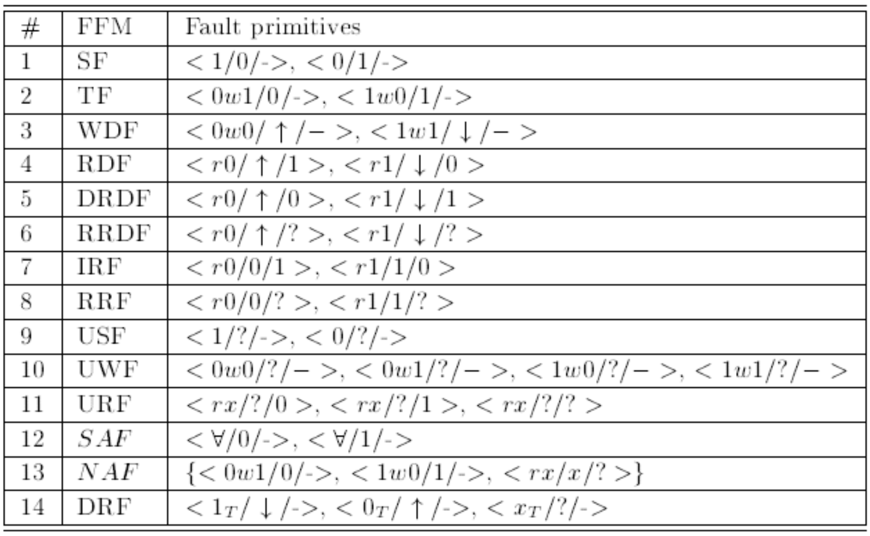
#	S	F	R	$<S/F/R>$	FFM	#	S	F	R	$<S/F/R>$	FFM
1	0	1	$-$	$<0/1/->$	SF	2	0	?	$-$	$<0/?/->$	USF
3	1	0	$-$	$<1/0/->$	SF	4	1	?	$-$	$<1/?/->$	USF
5	$0w0$	$\uparrow$	$-$	$<0w0/\uparrow/->$	WDF	6	$0w0$	?	$-$	$<0w0/?/->$	UWF
7	$1w1$	$\downarrow$	$-$	$<1w1/\downarrow/->$	WDF	8	$1w1$	?	$-$	$<1w1/?/->$	UWF
9	$0w1$	0	$-$	$<0w1/0/->$	TF	10	$0w1$	?	$-$	$<0w1/?/->$	UWF
11	$1w0$	1	$-$	$<1w0/1/->$	TF	12	$1w0$	?	$-$	$<1w0/?/->$	UWF
13	$r0$	0	1	$<r0/0/1>$	IRF	14	$r0$	0	?	$<r0/0/?>$	RRF
15	$r0$	$\uparrow$	0	$<r0/\uparrow/0>$	DRDF	16	$r0$	$\uparrow$	1	$<r0/\uparrow/1>$	RDF
17	$r0$	$\uparrow$	?	$<r0/\uparrow/?>$	RRDF	18	$r0$	?	0	$<r0/?/0>$	URF
19	$r0$	?	1	$<r0/?/1>$	URF	20	$r0$	?	?	$<r0/?/?>$	URF
21	$r1$	1	0	$<r1/1/0>$	IRF	22	$r1$	1	?	$<r1/1/?>$	RRF
23	$r1$	$\downarrow$	0	$<r1/\downarrow/0>$	RDF	24	$r1$	$\downarrow$	1	$<r1/\downarrow/1>$	DRDF
25	$r1$	$\downarrow$	?	$<r1/\downarrow/?>$	RRDF	26	$r1$	?	0	$<r1/?/0>$	URF
27	$r1$	?	1	$<r1/?/1>$	URF	28	$r1$	?	?	$<r1/?/?>$	URF

Since we have defined the *S, F* and *R* for 1PF1s, it is possible to list all FPs using this notation. Table 1 lists all possible combinations of the values, in the <*S/F/R* > notation, that result in FPs. The remaining combinations of the *S,* *F* and *R* values do not represent a faulty behavior. For example, <1w0/0/-> corresponds to a correct w0 operation after which the cell contains a 0, as expected. It is clear from Table 1 that the write operations are capable of sensitizing 8 FPs, and the read operations are capable of sensitizing 16 FPs. Note that in total there are 28 single-cell FPs.

2.3.2 Single-cell functional fault models

The list of 28 possible single-cell FPs will be compiled into a set of FFMs. The FFMs are given names, and each consists of a number of FPs. Selecting which FP should belong to a given generic FFM is rather arbitrary and is mainly determined by historical arguments; Table 3.2 summarizes the set of FFMs together with their FPs.

Table 2: List of all 1FP1s FFMs



#	FFM	Fault primitives
1	SF	$< 1/0/\text{-}>, < 0/1/\text{-}>$
2	TF	$< 0w1/0/\text{-}>, < 1w0/1/\text{-}>$
3	WDF	$< 0w0/\uparrow/- >, < 1w1/\downarrow/- >$
4	RDF	$< r0/\uparrow/1 >, < r1/\downarrow/0 >$
5	DRDF	$< r0/\uparrow/0 >, < r1/\downarrow/1 >$
6	RRDF	$< r0/\uparrow/? >, < r1/\downarrow/? >$
7	IRF	$< r0/0/1 >, < r1/1/0 >$
8	RRF	$< r0/0/? >, < r1/1/? >$
9	USF	$< 1/?/\text{-}>, < 0/?/\text{-}>$
10	UWF	$< 0w0/?/- >, < 0w1/?/- >, < 1w0/?/- >, < 1w1/?/- >$
11	URF	$< rx/?/0 >, < rx/?/1 >, < rx/?/? >$
12	SAF	$< \forall/0/\text{-}>, < \forall/1/\text{-}>$
13	NAF	$\{< 0w1/0/\text{-}>, < 1w0/1/\text{-}>, < rx/x/? >\}$
14	DRF	$< 1_T/\downarrow/\text{-}>, < 0_T/\uparrow/\text{-}>, < x_T/?/\text{-}>$

State Fault (SF): A cell is said to have a *state fault* if the logic value of the cell flips before it is accessed, even if no operation is performed on it. This fault is special in the sense that no operation is needed to sensitize it and, therefore, it only depends on the initial stored value in the cell.

Transition Fault (TF): A cell is said to have a *transition fault* if it fails to undergo a transition (‘0 to 1’ or ‘1 to 0’) when it is written. This FFM is sensitized by a write operation and depends on both the initial stored logic value and the type of the write operation.

Write Destructive Fault (WDF): A cell is said to have a *write destructive fault* if a non-transition write operation (0w0 or 1w1) causes a transition in the cell.

Read Destructive Fault (RDF): A cell is said to have a *read destructive fault* if a read operation performed on the cell changes the data in the cell, and returns an *incorrect* value on the output.

Deceptive Read Destructive Fault (DRDF): A cell is said to have a *deceptive read destructive fault* if a read operation performed on the cell returns the *correct* logic value, while it results in changing the contents of the cell.

Random Read Destructive Fault (RRDF): A cell is said to have a *random read destructive fault* if a read operation performed on the cell returns the *random* logic value, while it results in changing the contents of the cell.

Incorrect Read Fault (IRF): A cell is said to have an *incorrect read fault* if a read operation performed on the cell returns the incorrect logic value while keeping the correct stored value in the cell.

Random Read Fault (RRF): A cell is said to have a *random read fault* if a read operation performed on the cell returns a random data value on the output while the stored value remains as it is.

Undefined State Fault (USF): A cell is said to have an *undefined state fault* if the logic value of the cell flips to an undefined state before the cell is accessed, even if no operation is performed on it. This fault is special in the sense that no operation is needed to sensitize it and, therefore, it only depends on the initial stored value in the cell.

Undefined Write Fault (UWF): A cell is said to have an *undefined write fault* if the cell is brought in an undefined state by a write operation.

Undefined Read Fault (URF): A cell is said to have an *undefined read fault* if the cell is brought in an undefined state by a read operation. The returned data value during this read operation can be correct, wrong, or random.

Stuck-At Fault (SAF): A cell is said to have a *stuck-at fault* if it remains always stuck at a given value for all performed operations.

No Access Fault (NAF): A cell is said to have a *no access fault* if the cell is not accessible; i.e., the state of the cell cannot be changed with write operations, and any read operation applied to the cell returns a random data value.

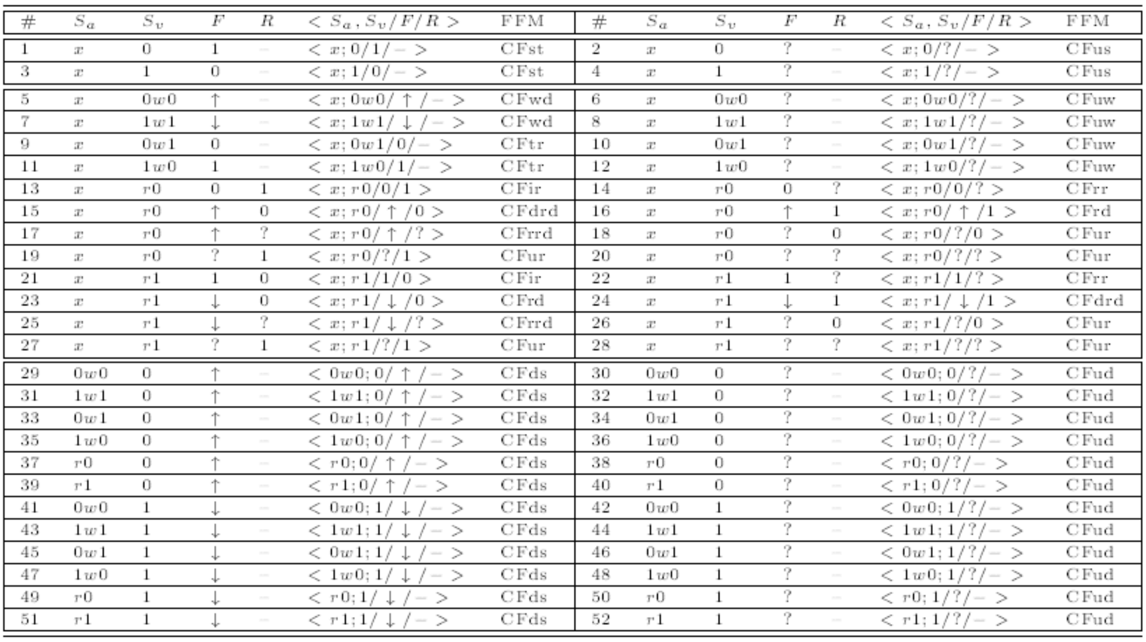
Data Retention Fault (DRF): A cell is said to have a *data retention fault* if the state of the cell changes after a certain time T, and without accessing the cell. *T* should be longer than the duration of the precharge cycle in SRAMs, because if the cell flips within the precharge cycle then the sensitized fault would be a state fault.

2.3.3 Two-cell fault primitives

Single-port FPs involving two cells (1PF2s) are divided into three types (Figure 13). Before listing the 1PF2s, a precise compact notation for 1PF2s, will be introduced.

$< S_a;S_v/F/R >$ (or $< S_a;S_v/F/R >_{a,v}$) denotes an FP involving two cells; S_a describes the sensitizing operation or state of the aggressor cell (a-cell); while S_v describes the sensitizing operation or state of the victim cell (v-cell). The a-cell (c_a) is the cell sensitizing a fault in another cell called the v-cell (c_v). The set Si is defined as: $Si \in$ {0,1, 0w0, 1w1, 0w1, 1w0, r0, r1} ($i \in$ {a, v}), $F \in$ {0,1, ↑, ↓,?}, and $R \in$ {0,1,?,-}.

Table 3: The complete set of 1PF2 FPs x ∈{0,1}



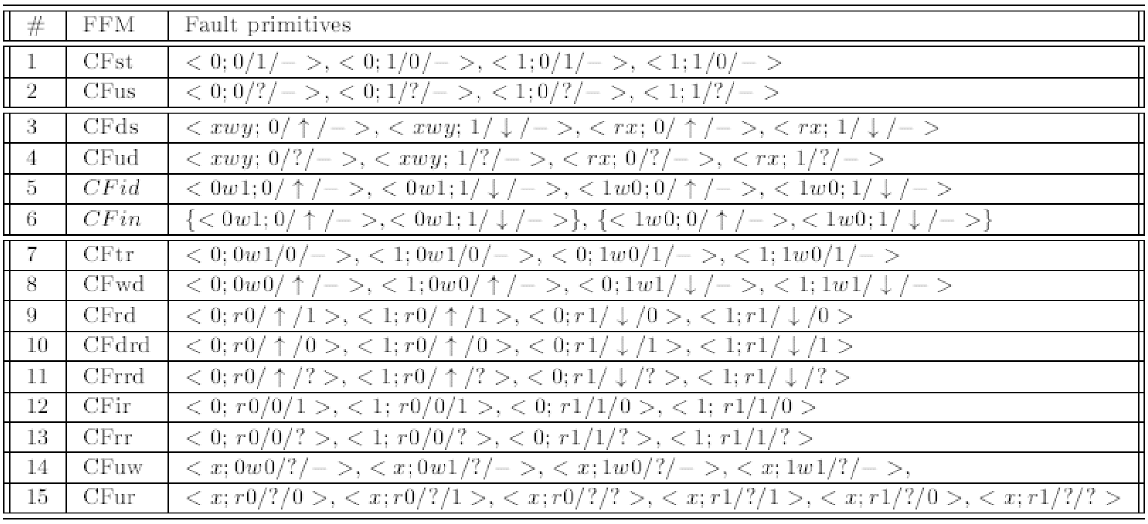
#	S_a	S_v	F	R	$< S_a, S_v/F/R >$	FFM	#	S_a	S_v	F	R	$< S_a, S_v/F/R >$	FFM
1	x	0	1	$-$	$< x; 0/1/- >$	CFst	2	x	0	?	$-$	$< x; 0/?/- >$	CFus
3	x	1	0	$-$	$< x; 1/0/- >$	CFst	4	x	1	?	$-$	$< x; 1/?/- >$	CFus
5	x	$0w0$	$\uparrow$	$-$	$< x; 0w0/\uparrow/- >$	CFwd	6	x	$0w0$	?	$-$	$< x; 0w0/?/- >$	CFuw
7	x	$1w1$	$\downarrow$	$-$	$< x; 1w1/\downarrow/- >$	CFwd	8	x	$1w1$	?	$-$	$< x; 1w1/?/- >$	CFuw
9	x	$0w1$	0	$-$	$< x; 0w1/0/- >$	CFtr	10	x	$0w1$	?	$-$	$< x; 0w1/?/- >$	CFuw
11	x	$1w0$	1	$-$	$< x; 1w0/1/- >$	CFtr	12	x	$1w0$	?	$-$	$< x; 1w0/?/- >$	CFuw
13	x	$r0$	0	1	$< x; r0/0/1 >$	CFir	14	x	$r0$	0	?	$< x; r0/0/? >$	CFrr
15	x	$r0$	$\uparrow$	0	$< x; r0/\uparrow/0 >$	CFdrd	16	x	$r0$	$\uparrow$	1	$< x; r0/\uparrow/1 >$	CFrd
17	x	$r0$	$\uparrow$	?	$< x; r0/\uparrow/? >$	CFrrd	18	x	$r0$	?	0	$< x; r0/?/0 >$	CFur
19	x	$r0$	?	1	$< x; r0/?/1 >$	CFur	20	x	$r0$	?	?	$< x; r0/?/? >$	CFur
21	x	$r1$	1	0	$< x; r1/1/0 >$	CFir	22	x	$r1$	1	?	$< x; r1/1/? >$	CFrr
23	x	$r1$	$\downarrow$	0	$< x; r1/\downarrow/0 >$	CFrd	24	x	$r1$	$\downarrow$	1	$< x; r1/\downarrow/1 >$	CFdrd
25	x	$r1$	$\downarrow$	?	$< x; r1/\downarrow/? >$	CFrrd	26	x	$r1$	?	0	$< x; r1/?/0 >$	CFur
27	x	$r1$	?	1	$< x; r1/?/1 >$	CFur	28	x	$r1$	?	?	$< x; r1/?/? >$	CFur
29	$0w0$	0	$\uparrow$	$-$	$< 0w0; 0/\uparrow/- >$	CFds	30	$0w0$	0	?	$-$	$< 0w0; 0/?/- >$	CFud
31	$1w1$	0	$\uparrow$	$-$	$< 1w1; 0/\uparrow/- >$	CFds	32	$1w1$	0	?	$-$	$< 1w1; 0/?/- >$	CFud
33	$0w1$	0	$\uparrow$	$-$	$< 0w1; 0/\uparrow/- >$	CFds	34	$0w1$	0	?	$-$	$< 0w1; 0/?/- >$	CFud
35	$1w0$	0	$\uparrow$	$-$	$< 1w0; 0/\uparrow/- >$	CFds	36	$1w0$	0	?	$-$	$< 1w0; 0/?/- >$	CFud
37	$r0$	0	$\uparrow$	$-$	$< r0; 0/\uparrow/- >$	CFds	38	$r0$	0	?	$-$	$< r0; 0/?/- >$	CFud
39	$r1$	0	$\uparrow$	$-$	$< r1; 0/\uparrow/- >$	CFds	40	$r1$	0	?	$-$	$< r1; 0/?/- >$	CFud
41	$0w0$	1	$\downarrow$	$-$	$< 0w0; 1/\downarrow/- >$	CFds	42	$0w0$	1	?	$-$	$< 0w0; 1/?/- >$	CFud
43	$1w1$	1	$\downarrow$	$-$	$< 1w1; 1/\downarrow/- >$	CFds	44	$1w1$	1	?	$-$	$< 1w1; 1/?/- >$	CFud
45	$0w1$	1	$\downarrow$	$-$	$< 0w1; 1/\downarrow/- >$	CFds	46	$0w1$	1	?	$-$	$< 0w1; 1/?/- >$	CFud
47	$1w0$	1	$\downarrow$	$-$	$< 1w0; 1/\downarrow/- >$	CFds	48	$1w0$	1	?	$-$	$< 1w0; 1/?/- >$	CFud
49	$r0$	1	$\downarrow$	$-$	$< r0; 1/\downarrow/- >$	CFds	50	$r0$	1	?	$-$	$< r0; 1/?/- >$	CFud
51	$r1$	1	$\downarrow$	$-$	$< r1; 1/\downarrow/- >$	CFds	52	$r1$	1	?	$-$	$< r1; 1/?/- >$	CFud

Table 3 lists all possible combinations of the values, in the $< S_a;S_v/F/R >$ notation, that result in FPs. The column ’FFM’ in the table shows the FFM each FP belongs to; such FFMs will be discussed in more detail in the next section.

2.3.4 Two-cell functional fault models

The list of 80 possible 1PF2 FPs will be compiled into a set of FFMs. Table 4 summarizes the set of FFMs together with their FPs; each of the FFM will be discussed in detail. Remember that the 1PF2 FPs are divided into three types $1PF2_s$, $1PF2_a$ and $1PF2_v$.

Table 4: List of 1PF2 FFMs x,y∈{0,1}



#	FFM	Fault primitives
1	CFst	$< 0; 0/1/- >, < 0; 1/0/- >, < 1; 0/1/- >, < 1; 1/0/- >$
2	CFus	$< 0; 0/?/- >, < 0; 1/?/- >, < 1; 0/?/- >, < 1; 1/?/- >$
3	CFds	$< xwy; 0/\uparrow/- >, < xwy; 1/\downarrow/- >, < rx; 0/\uparrow/- >, < rx; 1/\downarrow/- >$
4	CFud	$< xwy; 0/?/- >, < xwy; 1/?/- >, < rx; 0/?/- >, < rx; 1/?/- >$
5	*CFid*	$< 0w1; 0/\uparrow/- >, < 0w1; 1/\downarrow/- >, < 1w0; 0/\uparrow/- >, < 1w0; 1/\downarrow/- >$
6	*CFin*	$\{< 0w1; 0/\uparrow/- >, < 0w1; 1/\downarrow/- >\}, \{< 1w0; 0/\uparrow/- >, < 1w0; 1/\downarrow/- >\}$
7	CFtr	$< 0; 0w1/0/- >, < 1; 0w1/0/- >, < 0; 1w0/1/- >, < 1; 1w0/1/- >$
8	CFwd	$< 0; 0w0/\uparrow/- >, < 1; 0w0/\uparrow/- >, < 0; 1w1/\downarrow/- >, < 1; 1w1/\downarrow/- >$
9	CFrd	$< 0; r0/\uparrow/1 >, < 1; r0/\uparrow/1 >, < 0; r1/\downarrow/0 >, < 1; r1/\downarrow/0 >$
10	CFdrd	$< 0; r0/\uparrow/0 >, < 1; r0/\uparrow/0 >, < 0; r1/\downarrow/1 >, < 1; r1/\downarrow/1 >$
11	CFrrd	$< 0; r0/\uparrow/? >, < 1; r0/\uparrow/? >, < 0; r1/\downarrow/? >, < 1; r1/\downarrow/? >$
12	CFir	$< 0; r0/0/1 >, < 1; r0/0/1 >, < 0; r1/1/0 >, < 1; r1/1/0 >$
13	CFrr	$< 0; r0/0/? >, < 1; r0/0/? >, < 0; r1/1/? >, < 1; r1/1/? >$
14	CFuw	$< x; 0w0/?/- >, < x; 0w1/?/- >, < x; 1w0/?/- >, < x; 1w1/?/- >,$
15	CFur	$< x; r0/?/0 >, < x; r0/?/1 >, < x; r0/?/? >, < x; r1/?/1 >, < x; r1/?/0 >, < x; r1/?/? >$

The 1PF2$_s$ FFMs

This type has the property that the *state* of the a-cell, rather than an operation applied to the a-cell, sensitizes a fault in the v-cell; it consists of two FFMs:

State coupling fault (CFst): Two cells are said to have a *state coupling fault* if the v-cell is forced into a given logic state only if the a-cell is in a given state, without performing any operation on the v-cell or on the a-cell. This fault is special in the sense that no operation is needed to sensitize it and, therefore, it only depends on the initial stored values in the cells.

The 1PF2$_a$ FFMs

This type has the property that the application of a single-port operation to the a-cell sensitizes a fault in the v-cell; it consists of the following FFMs:

Disturb coupling fault (CFds): Two cells are said to have a *disturb coupling fault* if an operation (write or read) performed on the a-cell causes the v-cell to flip. Here, any operation performed on the a-cell is accepted as a sensitizing operation for the fault, be it a read, a transition write or a non-transition write operation.

Idempotent coupling fault (CFid): Two cells are said to have an *idempotent coupling fault* if a transition write operation (0w1 and 1w0) on the a-cell causes the v-cell to flip. This fault is sensitized by a *transition write* operation performed on the a-cell.

Inversion coupling fault (CFin): Two cells are said to have an *inversion coupling fault* if the logic value of the v-cell is inverted in case a *transition write* operation is performed on the a-cell.

The 1PF2$_v$ FFMs

This type has the property that the application of a single-port operation to the v-cell (with the a-cell in certain state) sensitizes a fault in the v-cell. It consists of the following FFMs:

Transition coupling fault (CFtr): Two cells are said to have a *transition coupling fault* if a given logic value in the aggressor results in a faulty transition write operation performed on the victim. This fault is sensitized by first setting the a-cell in a given state, and thereafter applying a write operation on the v-cell.

Write Destructive coupling fault (CFwd): Two cells are said to have a *write destructive coupling fault* if a non-transition write operation performed on the v-cell results in a transition when the a-cell is in a given logic state.

Read Destructive coupling fault (CFrd): Two cells are said to have a *read destructive coupling fault* when a read operation performed on the v-cell changes the data in the v-cell and returns an *incorrect* value on the output, if the a-cell is in a given state.

Deceptive Read Destructive coupling fault (CFdrd): Two cells are said to have a *deceptive read destructive coupling fault* when a read operation performed on the v-cell changes the data in the v-cell and returns a *correct* value on the output, if the a-cell is in a given state.

Random Read Destructive coupling fault (CFrrd): Two cells are said to have a *random read destructive coupling fault* when a read operation performed on the v-cell changes the data in the cell and returns a *random* value on the output, if the a-cell is in a given state.

Incorrect Read coupling fault (CFir): Two cells are said to have an *incorrect read coupling fault* if a read operation performed on the v-cell returns the incorrect logic value when the a-cell is in a given state. Note here that the state of the v-cell is not changed.

Random Read coupling fault (CFrr): Two cells are said to have a *random read coupling fault* if a read operation performed on the v-cell changes the data in the v-cell and returns a *correct* value on the output, when the a-cell is in a given state. Note that the state of the v-cell is not impacted; it remains in its correct value.

Undefined Write coupling fault (CFuw): Two cells are said to have an *undefined write coupling fault* if the v-cell is brought in an undefined state by a write operation performed on the v-cell, when the a-cell is in a given state.

Undefined Read coupling fault (CFur): Two cells are said to have an *undefined read coupling fault* if the v-cell is brought in an undefined state by a read operation performed on the v-cell, when the a-cell is in a given state.

An analysis of the defined FFMs shows that all introduced FFMs are necessary except the CFid and CFin. These two FFMs have been introduced for historical reasons.

2.4 March tests for SRAM memories

2.4.1 March test notations

A *March test* consists of a finite sequence of March elements [23]. A *March element* is a finite sequence of operations applied to every cell in the memory before proceeding to the next cell. The way one proceeds to the next cell is determined by the address order, which can be an increasing address order (e.g., increasing address from the cell 0 to the cell n-1), denoted by ↑ symbol, or a decreasing address order, denoted by ↓ symbol, and which is the exact inverse of the ↑ address order. When the address order is irrelevant, the symbol ↕ (i.e., ↑ or ↓) will be used.

An *operation* can consist of:

- w0: write 0 into a cell.
- w1: write 1 into a cell.
- r0: read a cell with expected value 0.
- r1: read a cell with expected value 1.

A complete March test is delimited by the bracket pair; while a March element is delimited by the '(...)' bracket pair. The March elements are separated by semicolons, and the operations within a March element are separated by commas. For example, the *MATS+* March test {↑(w0);↑(r0, w1);↓(r1,w0)} consists of the March elements ↑(w0), ↑(r0, w1) and ↓(r1,w0). Note that all operations of a March element are performed at a certain address, before proceeding to the next address.

2.4.2 March test for single port faults

March SS detects all the single-cell and the double cell FFMs that has been presented in Section 3.3. March SS is shown below. It has a test length of 22*N.* Let $M_{i,j}$ denote the j^{th} operation of March element M_i ; e.g., $M_{1,3}$ denotes the third operation (i.e., w0) of M_1.

$$\mathrm{MarchSS}:\left\{\begin{array}{l}\underbrace{\uparrow(W_0)}_{M_0},\underbrace{\uparrow(R_0,R_0,W_0,R_0,W_1)}_{M_1},\underbrace{\uparrow(R_1,R_1,W_1,R_1,W_0)}_{M_2},\\ \underbrace{\downarrow(R_0,R_0,W_0,R_0,W_1)}_{M_3},\underbrace{\downarrow(R_1,R_1,W_1,R_1,W_0)}_{M_4},\underbrace{\downarrow(R_0)}_{M_5}\end{array}\right\}$$

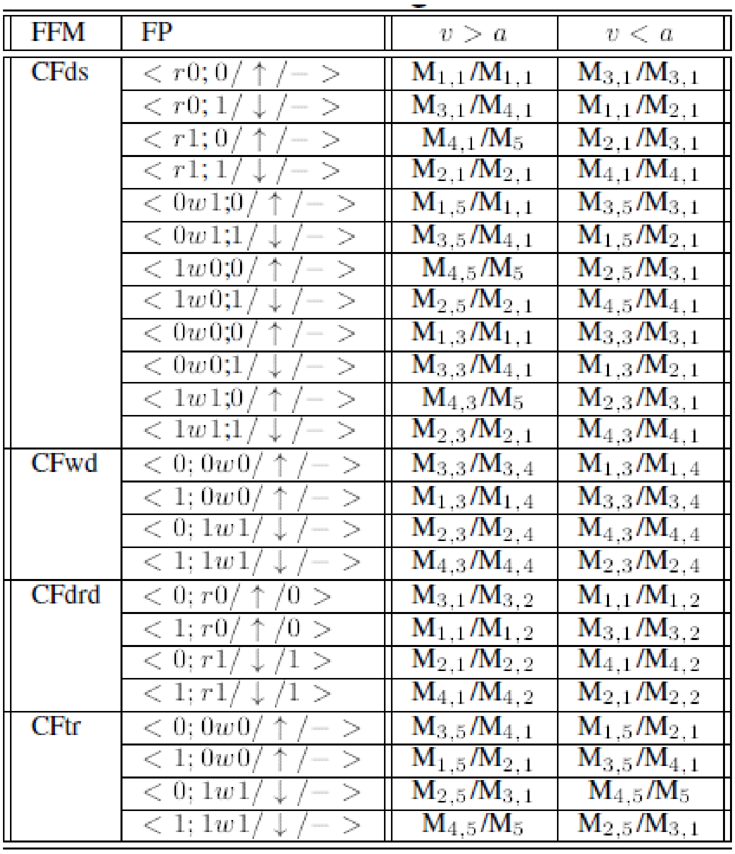
March SS detects all single-cell FFMs:

- All SFs, RDFs and IRFs are detected since from each cell a 0 and a 1 is read.
- All TFs are detected because each cell is read after an up and a down transition write operation. The <0w1/0/-> is sensitized by $M_{1,5}$ (also by $M_{3,5}$) and detected by $M_{2,1}$($M_{4,1}$); while the <1w0/0/-> is sensitized by $M_{2,5}$ (also by $M_{4,5}$) and detected by $M_{3,1}$(M_5).
- All WDFs are detected since each cell is read after a non-transition write operation; this is done by M_1 and M_2 (also by M_3 and M_4).
- All DRDFs are detected because two *successive* read operations are applied to each cell; the first read operation sensitizes the fault while the second detects it.

March SS detects all two-cell FFMs:

- The detection of CFst's requires that four states of any two cells can be generated and verified by a read operation. In March SS all states of any two cells c_i and c_j (i.e., 00, 01, 11, 10) are generated and verified (it can be easily verified by using a state diagram).
- All CFds's are detected; this include CFds's based on read operations, on transition write operations and on non-transition write operations. The first block of Table 5 shows by which March element (i.e., M_0 through M_5) of March SS, each FP belonging to each FFM is sensitized and detected. In the table, two cases have been distinguished: a) the v-cell has a higher address than the a-cell (i.e., *u* > *a),* and b) the v-cell has a lower address than the a-cell (u < *a).* In addition, in each entry the notation Sensitization/Detection is used. E.g., the < r0;0/↑/- > is sensitized and detected by $M_{1,1}$ when *u* > *a*; while < r0;1/↑/- > is sensitized by $M_{3,1}$ and detected by $M_{4,1}$ when *u* > *a*.

Table 5: March SS Fault coverage



FFM	FP	$v > a$	$v < a$
CFds	$< r0; 0/\uparrow/- >$	$M_{1,1}/M_{1,1}$	$M_{3,1}/M_{3,1}$
	$< r0; 1/\downarrow/- >$	$M_{3,1}/M_{4,1}$	$M_{1,1}/M_{2,1}$
	$< r1; 0/\uparrow/- >$	$M_{4,1}/M_{5}$	$M_{2,1}/M_{3,1}$
	$< r1; 1/\downarrow/- >$	$M_{2,1}/M_{2,1}$	$M_{4,1}/M_{4,1}$
	$< 0w1;0/\uparrow/- >$	$M_{1,5}/M_{1,1}$	$M_{3,5}/M_{3,1}$
	$< 0w1;1/\downarrow/- >$	$M_{3,5}/M_{4,1}$	$M_{1,5}/M_{2,1}$
	$< 1w0;0/\uparrow/- >$	$M_{4,5}/M_{5}$	$M_{2,5}/M_{3,1}$
	$< 1w0;1/\downarrow/- >$	$M_{2,5}/M_{2,1}$	$M_{4,5}/M_{4,1}$
	$< 0w0;0/\uparrow/- >$	$M_{1,3}/M_{1,1}$	$M_{3,3}/M_{3,1}$
	$< 0w0;1/\downarrow/- >$	$M_{3,3}/M_{4,1}$	$M_{1,3}/M_{2,1}$
	$< 1w1;0/\uparrow/- >$	$M_{4,3}/M_{5}$	$M_{2,3}/M_{3,1}$
	$< 1w1;1/\downarrow/- >$	$M_{2,3}/M_{2,1}$	$M_{4,3}/M_{4,1}$
CFwd	$< 0; 0w0/\uparrow/- >$	$M_{3,3}/M_{3,4}$	$M_{1,3}/M_{1,4}$
	$< 1; 0w0/\uparrow/- >$	$M_{1,3}/M_{1,4}$	$M_{3,3}/M_{3,4}$
	$< 0; 1w1/\downarrow/- >$	$M_{2,3}/M_{2,4}$	$M_{4,3}/M_{4,4}$
	$< 1; 1w1/\downarrow/- >$	$M_{4,3}/M_{4,4}$	$M_{2,3}/M_{2,4}$
CFdrd	$< 0; r0/\uparrow/0 >$	$M_{3,1}/M_{3,2}$	$M_{1,1}/M_{1,2}$
	$< 1; r0/\uparrow/0 >$	$M_{1,1}/M_{1,2}$	$M_{3,1}/M_{3,2}$
	$< 0; r1/\downarrow/1 >$	$M_{2,1}/M_{2,2}$	$M_{4,1}/M_{4,2}$
	$< 1; r1/\downarrow/1 >$	$M_{4,1}/M_{4,2}$	$M_{2,1}/M_{2,2}$
CFtr	$< 0; 0w0/\uparrow/- >$	$M_{3,5}/M_{4,1}$	$M_{1,5}/M_{2,1}$
	$< 1; 0w0/\uparrow/- >$	$M_{1,5}/M_{2,1}$	$M_{3,5}/M_{4,1}$
	$< 0; 1w1/\downarrow/- >$	$M_{2,5}/M_{3,1}$	$M_{4,5}/M_{5}$
	$< 1; 1w1/\downarrow/- >$	$M_{4,5}/M_{5}$	$M_{2,5}/M_{3,1}$

- All CFwd's are detected. The detection of CFwd's requires that each pair of cells undergoes the four states (00, 01, 10, 11), the application of a non-transition operation and thereafter a read operation. The second block of Table 5 shows by which March element each FP of CFwd is sensitized and detected.

- All CFdrd's, CFrd's, CFir's are detected. The detection of CFrd's and CFir's require that each pair of cells undergoes the four states (00, 01, 10, 11), and a read operation has to be performed to each of the two cell; while the detection of CFdrd's requires, in addition, the application of another read operation. Therefore, any test detecting CFdrd also detects CFrd and CFir. The third block of Table 5 shows by which March element each FP of CFdrd is sensitized and detected.

- All CFtr's are detected. The detection of CFtr's requires that each pair of cells undergoes the four states (00, 01, 10, 11), the application of a transition write operation to sensitize the fault, and thereafter a read operation to detect it. The fourth block of Table 5 shows by which March element each FP of CFtr is sensitized and detected.

2.5 CAM memories

2.5.1 CAM memories architecture

Content-addressable memories (CAMs) are a special type of memories used in high speed searching applications, e.g., in computer networking devices, processor caches, etc. The block diagram of a typical CAM is shown in Figure 14, where optional input–output signals and functional blocks are drawn by dashed lines—they may or may not be required, depending on the application.

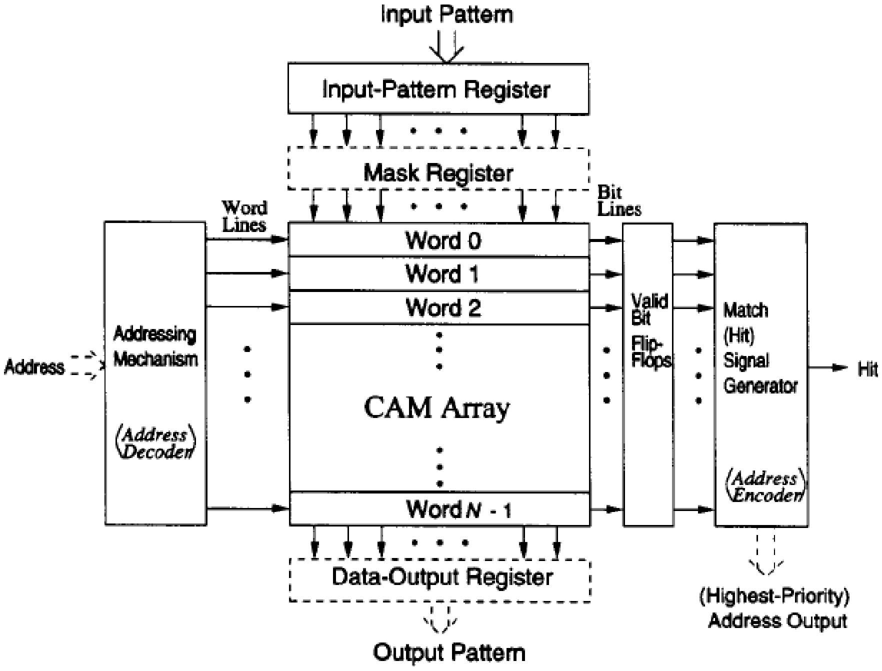


Figure 14: CAM memory block diagram

The implementation of some functional blocks, such as the address decoder and encoder, also depends on the requirements of the system containing the CAM. The searching (or matching) function of most CAM designs is performed by a *compare* operation, by which an *input pattern* (the input data word) is compared with all words stored in the CAM cell array simultaneously. There are only two possible comparison results for each word: *match* or *mismatch*. A match between a word and the input pattern means that all cells (bits) in the word are identical to the corresponding bits of the input pattern; otherwise, a mismatch occurs. A CAM is composed of *words*; a word is composed of *cells*; and a cell stores a single-bit value. A widely used SRAM-based single-bit CAM cell is shown in Figure 15a. This cell is substantially different from a traditional SRAM cell because it requires a larger number of transistors to perform the additional function of compare (as well as write and read). A CAM cell uses different bit lines for the read/write and compare operations and has a dual port structure, therefore, the concurrent operation (read & compare) is possible. A different implementation uses the same bit lines for read/write and compare (single or uni-port), as shown in Figure 15b; in this case, the concurrent operation (read & compare) is not possible (although the traditional read, write, and compare operations are still possible). The difference in functionality and the internal structure of the cell also affects the testing process of these devices [24][25].

The cell of an SRAM-based CAM can be divided into two functional parts: the storage part and the comparison part. The storage part (Figure 15) is the same as in a high density SRAM cell. It consists of six transistors and can be read and written through the read and write operations. The comparison part is unique to a CAM and consists of three transistors (denoted as M_0, M_1, and M_2).

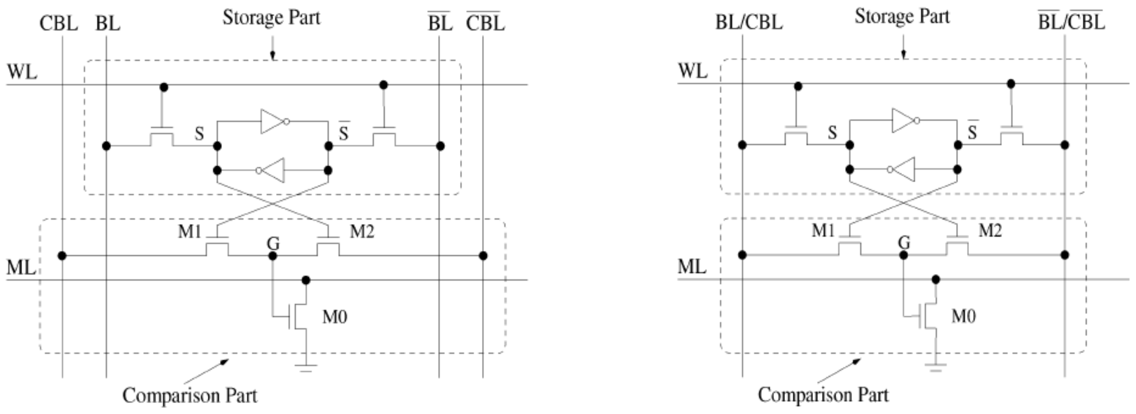


Figure 15: SRAM-based CAM cell: a) 2-port configuration b) uniport configuration

2.5.2 Faults for CAM memories

The storage part of a CAM cell is similar to an SRAM cell. Therefore all the functional fault models that were presented in section 2.3 are also mapped to the storage parts of the CAM memories. We will now describe the faults in detail that are modeled for the comparison part of the CAM memories.

The *stuck-matched fault (SMF)* is one that causes a CAM cell to always match its corresponding input bit irrespective of the state of the CAM cell and the input pattern. On the contrary, if there is a *stuck-mismatched fault (SMMF)* in a CAM cell, there will be no match for the cell irrespective of the state of the CAM cell and the input pattern.

Sometimes a defect may result in incorrect compare operations without affecting the normal write operations. For example, it is assumed that any write operation always writes zero into a cell under the BL–GND short. In such a case, the defect causes the cell to have an SMF. However, if both BL lines are at logic 0 during a write-1 operation, and the logic 0 on BL is weaker than that on $\overline{\text{BL}}$, then the write-1 operation can write a 1 into the cell successfully, or the write-1 operation does not change the cell value if the cell is already in the 1 state (maybe due to the power-on process). In other words, we cannot guarantee that the cell always stores a logic 0 after the write-1 operations. Such a defect will be mapped to another type of fault model, which will be discussed later in this section. To determine which logic-0 signal is stronger, the capability of the BL drivers and the physical characteristics of the short must be considered.

A CAM cell with a *conditional-match fault (CMF)* will function correctly if a logic value *x* has been written into it. However, the cell always provides an incorrect result for the subsequent compare operations if *x* has been written into it. The CMF can be further divided into the conditional-match-1 fault (CM1F) for *x=1* and the conditional-match-0 fault (CM0F) for *x=0*.

If there is a *partial-match fault (PMF)*, a CAM cell will be stuck-matched for all subsequent compare operations when a logic value *x* is written into the cell, and it will be stuck-mismatched when *x* is written into it. The PMF can be further classified as the partial-match-1 fault (PM1F) for *x=1* and the partial-match-0 fault (PM0F) for *x=0*.

There is a class of mismatch faults which we call the *equivalence-mismatch fault (EMMF)* - the compare operation fails if the CAM cell stores a value *x* and is compared with the same input value *x*. It is an equivalence-mismatch-1 fault (EMM1F) if *x=1*, and an equivalence-mismatch-0 fault (EMM0F) if *x=0*.

Similar to EMMF, there is another class of faults which we call the *inequivalence-match fault (IMF)* - the compare operation fails if the CAM cell stores a value *x* and is compared with the complementary input value *x*. It is an inequivalence-match-1 fault (IM1F) if *x=1*, and an inequivalence-match-0 fault (IM0F) if *x=0*.

A *cross-match fault (XMF)* or *cross-mismatch fault (XMMF)* is caused by a short between two neighboring BL and $\overline{\text{BL}}$ lines, which respectively belong to two neighboring cells in the same word. This defect will affect the match function of a word under certain input patterns. Let the two affected cells be denoted as $cell_j$ and $cell_{j+1}$, respectively. The BL line of $cell_j$ is shorted to the BL line of $cell_{j+1}$. If (11) is the input combination to be written into or compared with the two cells, both BL lines of the two cells are driven to VDD and both BL lines are pulled down to ground. Due to the short, the the BL line of $cell_{j+1}$ will be pulled up by the BL line of $cell_j$ to an intermediate voltage level between VDD and ground. If (11) is about to be written into the two cells, and we assume that the voltage difference between the bit lines of either cell is still large enough to force the cell to change its internal value, then the write operation is not affected, except that more power consumption and longer delay occurs during the write operation. On the other hand, if (11) is to be compared with the cells, there will be two possible results. First, if the intermediate voltage is high enough to turn M0 on, the *match* node of $cell_{j+1}$ is always low during the compare operation irrespective of the state of $cell_{j+1}$. Thus, there is an XMMF in $cell_{j+1}$. Second, if the intermediate voltage is not higher than the threshold voltage of M0, there is always a match for $cell_j$ after the compare operation, and the defect corresponds to an XMF.

When (00) is compared with the two cells, $cell_{j+1}$ always has a match under an XMMF, or $cell_j$ always has a mismatch under an XMF. For both XMMF and XMF, comparing (01) or (10) with the two involved cells will not cause malfunction.

2.6 March tests for CAM memories

To test a dual-port CAM, two different algorithms (concurrent and non-concurrent versions) are developed to account for the common implementations (dual-port and uni-port) [24].

2.6.1 The concurrent algorithm: CDA

In a concurrent March test algorithm that target CAM comparison faults, the concurrent *match* operations must be arranged carefully such that the relationship between the match input and the state of a cell can be fully characterized. Concurrent Detection Algorithm (CDA) detects all the above described modeled faults in the SRAM-based dual-port CAM of Figure 15a. The CDA algorithm is depicted in Figure 16.

If Passes 3 and 5 and all concurrent operations are removed, then CDA is reduced to the original March C algorithm, thus meeting the objective of full compatibility with existing test tools. As CDA is word-oriented, then all CAM cells in a single word are tested simultaneously. As the match line behaves as a wiring-AND (namely, the match lines will produce a mismatch output whenever at least a bit of the word is mismatched), then the match lines will generate as output a match only when all bits of the word are matched.

In CDA, Pass3 and Pass 5 utilize bit-based walking-0/1 patterns which are not encountered in the original March C algorithm; these passes are employed to detect some special faults, such as stuck-at-0 faults on compare bit lines. To detect stuck-at-0 faults in CBL/$\overline{\text{CBL}}$, the test pattern must produce a mismatch as fault-free output and a match as faulty output. Hence, the word-oriented data patterns (the all-0 or all-1 data patterns) used for cell states and comparison inputs cannot detect a stuck-at-0 in CBL/$\overline{\text{CBL}}$, i.e., the original March C algorithm with the added concurrent *match* operations in Passes 2, 4, 6, and 7 cannot detect the stuck-at-0 faults at CBL/$\overline{\text{CBL}}$. Therefore, CDA must employ bit-based walking-0/1 test patterns in Pass 3 and 5 to detect these faults. In Pass 3 or 5, only one bit of a word will be mismatched at a time and all other bits of the word are matched. So, if a CBL/$\overline{\text{CBL}}$ of a word is stuck-at-0, then the CAM will generate a match (instead of a mismatch) as output.

$Pass\ \ 1$	$\left[W\vert^{0}_{addr}\right]\Big\updownarrow^{N-1}_{addr=0}$
$Pass\ \ 2$	$\left[\begin{pmatrix} R\vert^{0}_{addr} \\ C\vert^{0} \end{pmatrix};(W\vert^{1}_{addr})\right]\Big\uparrow^{N-1}_{addr=0}$
$Pass\ \ 3$	$\left[C\vert^{2^{L}-2^{k}-1}\right]\Big\uparrow^{L-1}_{k=0}$
$Pass\ \ 4$	$\left[\begin{pmatrix} R\vert^{1}_{addr} \\ C\vert^{1} \end{pmatrix};(W\vert^{0}_{addr})\right]\Big\uparrow^{N-1}_{addr=0}$
$Pass\ \ 5$	$\left[C\vert^{2^{k}}\right]\Big\uparrow^{L-1}_{k=0}$
$Pass\ \ 6$	$\left[\begin{pmatrix} R\vert^{0}_{addr} \\ C\vert^{1} \end{pmatrix};(W\vert^{1}_{addr})\right]\Big\downarrow^{addr=N-1}_{0}$
$Pass\ \ 7$	$\left[\begin{pmatrix} R\vert^{1}_{addr} \\ C\vert^{0} \end{pmatrix};(W\vert^{0}_{addr})\right]\Big\downarrow^{addr=N-1}_{0}$
$Pass\ \ 8$	$\left[R\vert^{0}_{addr}\right]\Big\updownarrow^{N-1}_{addr=0}$

Figure 16: Concurrent Detection Algorithm (CDA) for CAM memories

Equivalently, when the test patterns for detecting the stuck-at-1 fault at CBL/$\overline{\text{CBL}}$ are applied, the expected fault-free output is a match, i.e., all bits in the word are matched. So, if any CBL/$\overline{\text{CBL}}$ is stuck-at-1, then the output will be a mismatch; therefore, all CBL/$\overline{\text{CBL}}$ stuck-at-1 faults are tested and detected in parallel. The features of CDA can be described in detail as follows:

- Pass 1 initializes the state of all cells to 0.
- Storage faults are detected by the *read* and *write* operations in Passes 2, 4, 6, 7, and 8. The analysis is the same as in the original March C algorithm.
- Stuck-at-0 faults at CBL are detected in Pass 5. As the cells in the CAM have a 0 as data (written in Pass 4), so the walking-1 patterns at the comparison inputs detect stuck-at-0 faults in CBL. If CBL is fault-free, then the CAM reports a mismatch; otherwise, it reports the address of the faulty cell as the matched address.
- $\overline{\text{CBL}}$ stuck-at-0 faults are detected in Pass3 (same as for stuck-at-0 faults at CBL).
- Stuck-at-1 faults at CBL are detected in Pass 2 .If no $\overline{\text{CBL}}$ stuck-at-1 fault exists, then the CAM reports the matched address (except the first *match* operation in Pass 7). If a CBL stuck-at-1 fault occurs, then the CAM always reports that there is no matched address. Pass 7 also detects this type of fault (by the same reasons as previously described).
- Passes 4 and 6 detect stuck-at-1 faults at $\overline{\text{CBL}}$ (by the same reasons as above).
- Passes 2 and 4 detect ML stuck-at-0 and M_0 stuck-on faults which cause the CAM to report a wrong matched address.
- Both Passes 3 and 5 detect ML stuck-at-1 and M_0 stuck-open faults, which cause the CAM to report a matched address (instead of a no matched address). The difference between these two faults is that an ML stuck-at-1 fault causes the CAM to always wrongly report a matched address at each pass, while the M_0 stuck-open fault causes the CAM to wrongly report a match in only one pass. Note that the first *match* operation in Passes 4 and 6 can also detect ML stuck-at-1 faults.
- If the conflict in data values at G behaves as a wiring-AND short, then Pass 3 will detect M_1 stuck-on faults; otherwise (wiring-OR), these faults are detected by Pass 4.
- If the floating point G is always low (or retains its previous voltage), then Pass 5 detects M_1 stuck-open faults. Else (i.e., G is always high), Pass 2 will detect them. Usually, M_1 stuck-open faults are likely to cause the floating point G to be low or retain its previous value. Under this assumption, G is more likely to be 0 when Pass 5 is applied because, when the state of a faulty cell is changed from 1 to 0 by *write* in Pass 4, G is connected to $\overline{\text{CBL}}$ (which is in the nop state), i.e., state 0. Therefore,

G is more likely to be in state 0 after it is isolated by both the off transistors M_1 and M_2

- If the conflict in data values at G behaves as a wiring-AND short, then Pass 5 will detect M_2 stuck-on faults; else (wiring-OR), these faults are detected in Pass 2.
- If the floating point G is always low (or retains its previous state too), then Pass3 detects all M_2 stuck-open faults; else (i.e., G is always high), Pass 4 will detect them. As in a previous case, M_2 stuck-open faults more likely will cause G to retain its previous voltage. As the previous voltage of G before Pass 3 is 0, so Pass 3 will detect all M2 stuck-open faults.
- If the bridge fault between word and match lines is a wiring-OR short, then the faulty word line will always be high during the *write* operation of CDA. When *write* tries to write new data (with valued) to an address, then the data at the faulty address is also changed to the new data (i.e., d instead of d). This is detected by the *read* operations in CDA. These faults are detected by CDA in Passes 2, 4, 6 and 7 . If the bridge is a wiring-AND short, then the faults are detected by the first *read* operations of Passes 6 and 7 because this fault causes the address not to be accessed.
- Bridge faults between read/write and compare bit lines are detected in Passes 6 and 7. These faults cause the *read* operations in Passes 6 and 7 to always read d as output pattern (instead of the expected pattern d). These faults also cause the *match* operations in Passes 6 and 7 to report a wrong matched address.

As per the above analysis, CDA detects all modeled faults inclusive of storage faults, comparison faults, and faults across the storage and comparison parts of the CAM. CDA requires eight passes and (10N + 2 L) tests, where N is the number of words in the CAM and L is the word width.

2.6.2 The non-concurrent algorithm: NCDA

When concurrent operations are not allowed in a CAM, CDA must be modified as a non-concurrent algorithm while preserving the same testing capabilities and fault coverage. The non-concurrent CAM detection algorithm (NCDA) is depicted in Figure 17.

$$
\begin{array}{|ll|c|}
Pass & 1 & \left[W|^{0}_{addr}\right]\Big\updownarrow^{N-1}_{addr=0} \\
Pass & 2 & \left[(R|^{0}_{addr});(W|^{1}_{addr})\right]\Big|^{\uparrow N-1}_{addr=0} \\
Pass & 3 & \left[C|^{2^{L}-2^{k}-1}\right]\Big|^{\uparrow L-1}_{k=0} \\
Pass & 4 & \left[(R|^{1}_{addr});(W|^{0}_{addr})\right]\Big|^{\uparrow N-1}_{addr=0} \\
Pass & 5 & \left[C|^{2^{k}}\right]\Big|^{\uparrow L-1}_{k=0} \\
Pass & 6 & \left[(R|^{0}_{addr};(W|^{1}_{addr});(C|^{1})\right]\Big|^{addr=N-1}_{\downarrow 0} \\
Pass & 7 & \left[(R|^{1}_{addr});(W|^{0}_{addr});(C|^{0})\right]\Big|^{addr=N-1}_{\downarrow 0} \\
Pass & 8 & \left[R|^{0}_{addr}\right]\Big\updownarrow^{N-1}_{addr=0}
\end{array}
$$

Figure 17: Non concurrent detection algorithm (NCDA) for CAM memories

NCDA differs from CDA in Passes 2, 4, 6, and 7; in particular, the following features must be pointed out:

- Pass 1 initializes the state of all cells to 0.
- As previously, storage faults are detected by the *read* and *write* operations in Passes 2, 4, 6, 7 and 8.
- CBL stuck-at-0 faults are detected in Pass 5.
- $\overline{\text{CBL}}$ stuck-at-0 faults are detected in Pass 3.
- CBL stuck-at-1 faults are detected in Pass 7.
- Pass 6 detects $\overline{\text{CBL}}$ stuck-at-1 faults.
- Passes 6 and 7 detect ML stuck-at-0 and M_0 stuck-on faults.
- Both Passes 3 and 5 detect ML stuck-at-1 and M_0 stuck-open faults.
- Pass 3 (6) detects M_1 stuck-on faults if the conflict in data values at G is a wiring-AND (OR) short.
- If G is always low (high) or retains its previous value, then Pass 5 (7) detects M_1 stuck-open faults.
- If the data conflict at G behaves as a wiring-AND (OR) short, then Pass 5 (7) detects M_2 stuck-on faults.

- If G is always low (high) (or retaining its previous value), then Pass 3 (6) detects M_2 stuck-open faults.
- If the bridge between word and match lines is a wiring-OR (AND) short, then the fault is sensitized by the *write* operations and detected by the *read* (*match*) operations of NCDA. Note that, for the wiring-AND short, detection occurs in Passes 6 and 7 because this fault causes the match line to be stuck-at-0.
- Bridge faults between read/write bit lines and compare bit lines are detected by the *read* operations of NCDA.

It is easy to prove that NCDA is the non-concurrent version of CDA. Therefore, NCDA detects all the modeled faults with 100% coverage; however, due to the absence of concurrent operations, NCDA require (12N + 2 L) tests, i.e., an increase of 2N tests compared with CDA.

2.7 Memory Built-In Self-Test (MBIST)

There are two main industrial approaches for testing embedded memories of SOCs and microprocessors: external testing by direct access using *automatic test equipment* (ATE) and internal testing using Memory BIST. On the one hand, direct access to the embedded memory cores from the limited number of I/O pins needs a high-performance ATE, as well as very long testing time since tester channels are time-shared by different memories under test. Thus, external testing becomes infeasible, in particular for large VLSI devices where transistor to pin ratio is high. On the other hand, Memory BIST provides at-speed and high-bandwidth access to the embedded memories and it only needs a low cost ATE to initialize the test sessions during manufacturing testing or a system call during on-line testing and a mechanism to inspect the final results. However, although Memory BIST is state-of-the-art technology and the industry standard for embedded memory testing, unless carefully designed, it may induce excessive power, in addition to performance and area overhead, since embedded memories nowadays dominates the silicon area in modern microprocessors and SOCs.

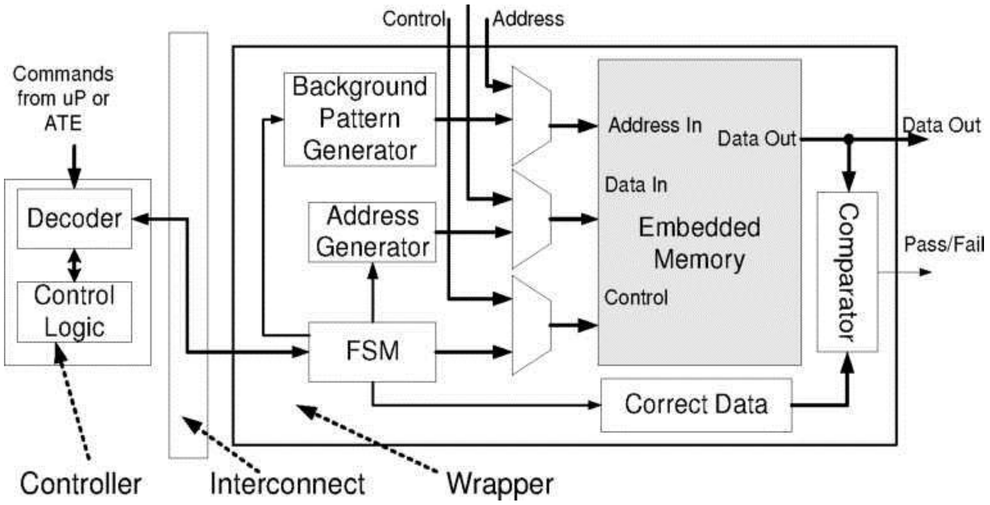


Figure 18: Generic Memory BIST architecture

A typical embedded MBIST approach comprises an MBIST wrapper, an MBIST controller and the interconnect between them, as shown in Figure 18. The MBIST wrapper further includes an address generator to provide complete memory address sequences (i.e., for n address lines all the 2n locations are visited in a complete sequence); a background pattern generator to produce data backgrounds when testing word-oriented memories; a comparator to check the memory output against the expected correct data pattern; and a finite state machine (FSM) to generate proper test control signals based on the commands received from the MBIST controller. The MBIST controller pre-processes the commands received from upper-level controller (either on-chip microprocessor or off-chip ATE) and then sends them to the MBIST wrapper. The interconnect between the wrapper and the controller could be either serial (i.e., a single command line is shared by all the wrappers) or parallel (i.e., dedicated multiple command lines are linking different wrappers to the controller).

The increasing size and number of embedded memories and the rapid development in VLSI process technologies lead to unique requirements for embedded memory BIST schemes when compares with other BIST mechanisms (e.g Pseudorandom Logic BIST):

- *Support multiple test algorithms*: The conventional MBIST approaches usually implement a single March test algorithm. However, deep submicron process technologies and design rules introduce physical defects that are not screened when using the memory test algorithms developed for previous process generations. Therefore MBIST architectures should be programmable to support multiple memory test algorithms to increase the fault coverage and to find the most suitable algorithms for the manufacturing process at hand.

- *Diagnosis and repair support*: Diagnosis support in MBIST architectures is mandatory for manufacturing yield enhancement for new process technology and a rapid transition from the yield ramp phase to the volume production phase. Furthermore, since embedded memories are subject to more aggressive design rules, they are more prone to manufacturing defects (caused by process variations) than logic in microprocessors and SOCs. For large embedded memories, the manufacturing yield can be unacceptable low. Hence, to achieve a certain manufacturing yield, in addition to diagnosis support, it is also beneficial to introduce self-repair features comprising redundant memory cells.

- *Test heterogeneous memories*: State-of-the-art microprocessors and SOCs include many types of memory cores, such as, among others, SRAM, DRAM, flash and ROM. Traditional MBIST approaches were designed to test only one type of memory. However, to reduce area and routing overhead via hardware resource sharing, as well as to decrease the testing time, it is advantageous to develop MBIST architectures that support testing heterogeneous memories simultaneously.

- *Power dissipation constraints*: For each memory the power dissipation will be identical in both test mode and normal functional mode since memory testing is functional testing. Therefore, if all memory blocks in a microprocessor or a SOC can be activated simultaneously during functional mode, power dissipation will not exceed the maximum power constraint during test. Hence, no test scheduling is required in this case. However, to reduce the overall testing time, test scheduling is still necessary for memory testing. On the one hand, for bus-connected memories (BCMs) which are connected to a single-master bus architecture, only one BCM can be accessed at any time during functional mode. If all BCMs are wrapped, then all of them can be activated simultaneously during test. Consequently, the power dissipation will be higher during test than during functional operation, and therefore, test scheduling is necessary. On the other hand, memory testing is part of SOC testing. Cores which use scan-based test methodology will consume more power during test than during functional mode. If the testing time of these scan-based cores is longer than that of memory cores, then by relaxing the power constraints for scan-based core testing and carefully scheduling memory testing with tightened power constraints, the overall testing time for the SOC can be reduced. Since test scheduling under power constraints is highly interrelated to the resource sharing mechanisms used in the MBIST architecture, it is essential to develop new power-constrained test scheduling algorithms that will get the maximum usage of the available hardware resources for embedded memory testing.

- *Reuse the available on-chip processing/communication resources*: Although all the embedded memory cores can be tested by adding dedicated memory BIST wrappers, the high area overhead of BIST circuitry, as well as the performance penalty caused by intrusive Design for Testability (DFT) hardware may prove to be the main drawback of this approach. Therefore, reusing the available on-chip resources for testing the embedded memories can lower the area and performance overhead associated with a high number of dedicated MBIST wrappers for BCMs. Furthermore, by implementing non-time-critical tasks in software using a processor, the complexity of the controller can also be reduced.

The objective of memory BIST approaches is to meet some or all of the above requirements while reducing the cost of testing by targeting low area and performance penalty and low testing time. The existing approaches have explored three main directions to gain improvements: memory BIST architectures, test scheduling algorithms, and special design implementations. Due to their interrelation, without a good architectural support it is hardly possible to achieve any significant improvements through test scheduling or special design techniques. Therefore, research approaches in the literature mainly focus on introducing new MBIST architectures.

A *memory BIST architecture* is defined by the integration of its three components shown in Figure 18 (controller, wrapper and interconnect). A *standalone* approach uses a dedicated wrapper and controller for each memory core (or memory cluster with several identical memory cores), while a *distributed* approach shares one controller to manage some or all of the MBIST wrappers in a microprocessor of a SOC.

In a *standalone MBIST* architecture, the BIST controller and the wrapper are physically close located, hence parallel interconnect between them can be used. The MBIST approach of each memory is independent of the other memories' BIST approaches, which makes the implementation of this approach straightforward.

MBIST approaches which support multiple March test algorithms are called programmable MBIST architectures. To support multiple March test algorithms, one can either implement all the March primitives or several March elements. Since there are only four March primitives (r0, w0, r1, w1), by implementing all of them with different combinations of background patterns and address sequences, any March algorithm can be supported. One programmable MBIST approach using March primitives was investigated in [26] and it includes an instruction memory to store the test instructions and a decoding logic to process the test instructions. March element-based approaches implement only several most commonly used March elements. Based on the implemented March elements, only a limited number of March algorithms can be supported. However, its main advantage lies in less area overhead (simpler decoding logic and less test instructions) when compared to March primitive-based approaches. In addition, by carefully selecting the March elements, new March test algorithms can be generated to target memory faults in new process technologies. A programmable FSM- based MBIST architecture with 7 March elements was researched in [26]. Another March element-based approach, which supports 40 March algorithms, was presented in [27]. However, both approaches use dedicated on-chip memory to store the test instructions, thus leading to large test area overhead. Furthermore, dedicated control signals are needed for each MBIST core, which may cause routing and test integration problems when the SOC comprises hundreds of memory cores. To overcome the control problem, a P1500-based [28] programmable MBIST architecture using March elements was introduced in [29]. Using P1500 core wrappers, the test controller (ATE or on-chip processing element) has the full controllability of all the wrapped memories and can send different test instructions to each MBIST wrapper during the test, thus eliminating also the need of an on-chip instruction memory. Note, however, if the SOC consists of a high number of embedded memory cores, and all of them are wrapped with fully-compliant P1500 wrappers, the main limitation is caused by the excessive wrapper area overhead and unnecessary performance degradation.

Diagnosis support is another important feature of MBIST architectures. A built- in self-diagnosis (BISD) scheme was introduced in [30]. It sends out faulty memory cell information (such as faulty address, data, and test session number) for failure analysis. To reduce the control complexity of this approach when testing numerous memory cores, a P1500 MBIST approach with diagnosis enhancement was proposed in [31]. To reduce the testing time in the diagnosis mode (caused by the serial scan-chain structure required to shift out the diagnosis information), a test response compression method was introduced in [32]. Using this method, less I/O pins can be used to send out the faulty response data compared with the uncompressed parallel solution. Due to the increased size of embedded memories, support for memory self-repair is becoming necessary to increase the overall SOC yield. Using the detailed location and information of faulty memory cells (provided by diagnosis support approaches discussed above), one can perform memory redundancy allocation and use fuse-boxes or other methods to repair the faulty memories. However, to collect enough information on fault locations for various memory faults, more complex March test algorithms are required, which implies longer testing time. An MBIST solution was introduced in [33] to test and repair large embedded DRAMs using on-chip redundancy allocation. To reduce the testing time, a memory BIST architecture was proposed in [34] with revised March test algorithms. While most of the previously described MBIST approaches are focused on testing single port SRAMs, as long as the test algorithms have the features of March algorithms, they are suitable for testing other types of memories with minor modifications. For example, a flash memory BIST architecture was proposed in [35] using a March-like test algorithm.

In summary, with the exception of the P1500 memory BIST approaches, most of the standalone MBIST architectures focus only on solving the test problems related to a single memory core or a standalone memory chip. They do not account for the specific requirements for integrating the design for test hardware for hundreds of embedded memory cores. They also do not provide any support for test scheduling under power dissipation constraints, which needs a flexible control mechanism for the memory BIST hardware. Although P1500 memory BIST approaches can solve the control problem, a fully-compliant P1500 wrapper and standalone MBIST hardware for all the embedded cores will introduce excessive area overhead and unnecessary performance degradation. To overcome these issues, a new system perspective for memory BIST architectures for complex SOCs is needed. The result turns out to be the distributed MBIST architecture and hardware/software co-testing solutions.

In a *distributed MBIST* architecture, each memory core still has a dedicated technology-dependent wrapper. However, depending on the complexity of the SOC, there are only one (or a few) BIST controllers used to direct the test of all the embedded memory cores. Since hardware resource sharing is introduced, to reduce the routing congestion and to facilitate rapid power-constrained testing, the interconnect between the wrappers and the controller(s) must be carefully considered.

Distributed BIST architectures have been advocated for over a decade. In [36], Zorian presented a distributed BIST control scheme to test the building blocks of a complex VLSI circuit. Due to the increasing ratio of the memory area in a state-of-the-art SOC, dedicated memory BIST architectures can be used to reduce the cost of memory testing. Distributed MBIST architectures can further be divided into: *hardware-centric*, *software-centric*, and *hardware/software (HW/SW) co-testing*.

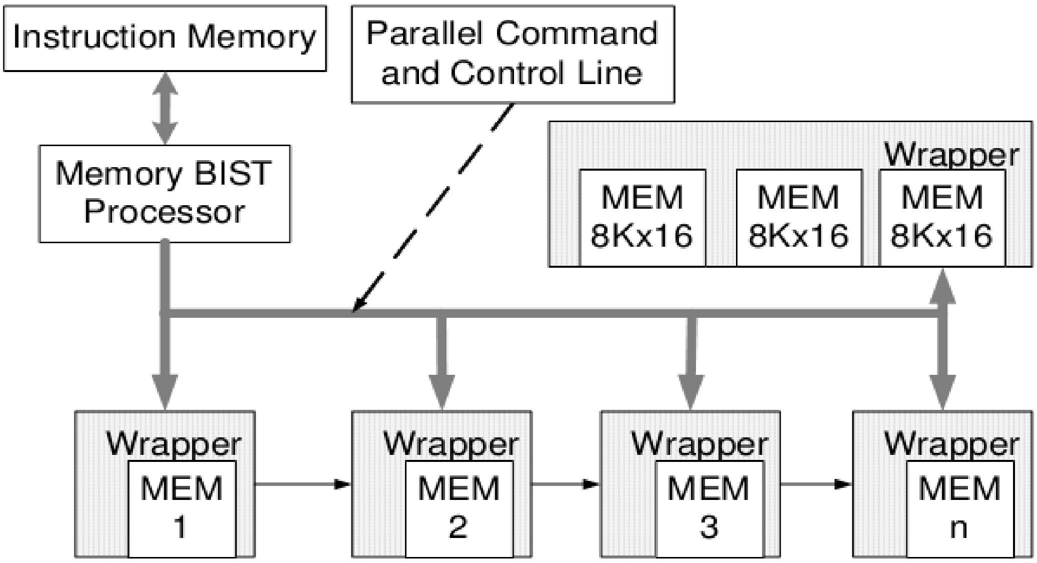


Figure 19: Distributed MBIST architecture

- *Hardware-centric MBIST architecture*: A hardware-centric approach uses dedicated hardware to test all the memory cores in a SOC or a microprocessor. It can achieve the near optimum testing time as well as supports flexible test scheduling. However, this approach also introduces large area overhead. A typical distributed hardware-centric MBIST architecture was proposed in [37] and is shown in Figure 19. Each memory (or memory cluster for several identical memories) has a dedicated technology-dependent wrapper. By extracting some technology independent tasks and the test instruction memory to a central controller, which controls all the wrappers, the overall BIST area overhead is reduced. This architecture also integrates several advanced features which have appeared previously in various standalone MBIST approaches. For example, the wrapper can run separate March primitive operations (e.g., r0 or w1) received from the controller. This implies that the hardware-centric MBIST architecture is programmable and supports multiple March algorithms. Besides, the wrapper design also supports diagnosis by scanning out the faulty addresses and background patterns. However, its main drawback lies in the interconnect between controller and wrappers, which uses one parallel command line to configure all the memory BIST wrappers to run the same test commands (for example, March primitives in this approach). This implies that for large SOCs, different types of memories (or memories requiring different test algorithms) cannot be tested simultaneously using the same BIST controller, thus increasing testing time as well as test control complexity. Moreover, using parallel interconnects between the controller and the wrappers, the routing congestion may become a potential problem when hundreds of embedded memory cores are present. Furthermore, the testing time for each test session is dominated by the largest memory, which may lead to prohibitively long testing time under power dissipation constraints.

- *Software-centric MBIST architecture*: A software-centric approach reuses the existing on-chip resources to test all the bus-connected memories. Since SOCs usually contain one or more processing elements, which use on-chip communication architectures to transfer data to/from some of the embedded cores. Reusing these resources for testing the bus-connected memories can lower the area overhead and eliminate the performance penalty caused by the MBIST wrappers. In [38], a methodology for testing SOCs using an on-chip microprocessor was presented. However, this approach uses only software to generate, analyze and apply the test algorithms for the bus-connected memories, which requires a much longer testing time than the existing hardware-centric approaches. This is because the hardware architecture can generate March algorithms more efficiently than software. Furthermore, it is obvious that without additional hardware support, the software-centric approaches can only test the bus-connected memories.

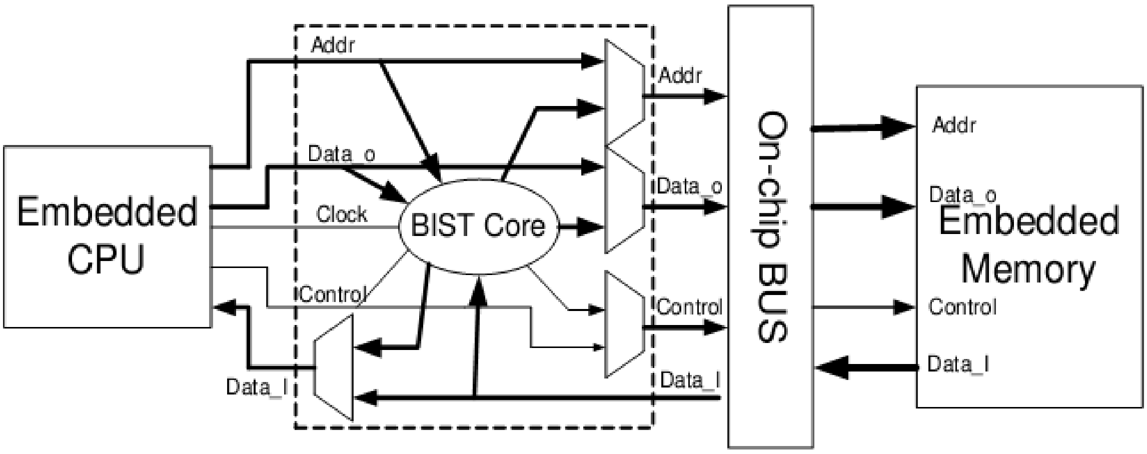


Figure 20: Memory BIST for bus-connected memories

- *Hardware/Software co-testing MBIST architecture*: This architecture takes advantage of both hardware-centric and software-centric approaches. By migrating all the non-time-critical tasks from the MBIST controller to the processor, such as fetching and decoding test instructions, one can reduce the area overhead with a minor testing time penalty. A processor-programmable memory BIST solution was proposed in [39], where a BIST circuit was inserted between the embedded central processing unit (CPU) and the system bus (Figure 20). Although it reduces the testing time problem associated with the software-centric approach, this solution may affect the overall SOC performance, since the BIST circuitry introduces extra multiplexers between the CPU and the bus, thus increasing the CPU access time to the bus. Moreover, this approach can only test bus-connected memories (BCMs), which is not a complete solution for SOC memory testing.

3. MICROPROCESSOR SOFTWARE-BASED SELF-TEST

Software-based self-test (SBST), also called instruction-based self-test, is the process of detecting physical defects (or faults that model them) in a processor or processor-based system by executing processor instructions in its normal mode of operation. Software-Based Self-Test (SBST) has recently emerged as an effective complementary solution for microprocessor and embedded processor manufacturing [40], as well as, periodic on-line testing [41] along with other components in Systems-on-Chip (SoCs). Key microprocessor companies (Sun [42], Intel [43]) have recognized the potential of SBST adopting it in their test flows. Recently, in [44], a taxonomy for different SBST methodologies has been presented. SBST is a non-intrusive approach that embeds a “software tester” with the form of a self-test program in the processor’s on-chip memory. This way SBST imposes zero hardware and performance overhead during normal operation, as well as, ordinary power density during on-line testing. It leverages the use of low-speed, reduced pin-count external ATE providing high quality, at-speed testing virtually without introducing any hardware or performance overheads. Moreover, SBST can be easily reused in field for power up diagnostics or periodic on-line testing to add dependability features.

An outline of the software-based self-test concept is shown in Figure 21. We provide an elaboration on all recent approaches in this chapter.

Initially, the self-test program is loaded into the processor’s on-chip memory using a low-speed, low-cost *structural tester* (Figure 21a). Secondly, during test application (Figure 21b), the processor executes the self-test program from its on-chip memory at its normal clock frequency, thus achieving full at-speed testing. During this phase, the processor collects the test responses (possibly compressed in a test signature), and stores them in its on-chip data memory (Figure 21c). Finally, the low-speed, low-cost tester is used again to unload the test responses from the on-chip memory for further external analysis (Figure 21d). Since modern microprocessors integrate large multilevel caches on the same die, execution from on-chip cache is considered a further advantage provided that a cache-loader mechanism exists to load the test program and unload the test response(s).

SBST changes the role of the external ATE from actual test application to a simple interface with the on-chip memory before and after the test execution. Therefore, SBST achieves the goal of at-speed testing using low-speed ATE. In addition, since the means for applying SBST programs are existing processor instructions, at-speed testing is feasible without the risk of thermal damage due to excessive signal activity in special test modes of circuit operation. Furthermore, by utilizing the processor's Instruction Set Architecture (ISA) and complying with all the restrictions enforced by both the ISA and the designers' decisions, SBST avoids *overtesting* (for faults that do not appear during normal circuit operation) and saves valuable yield.

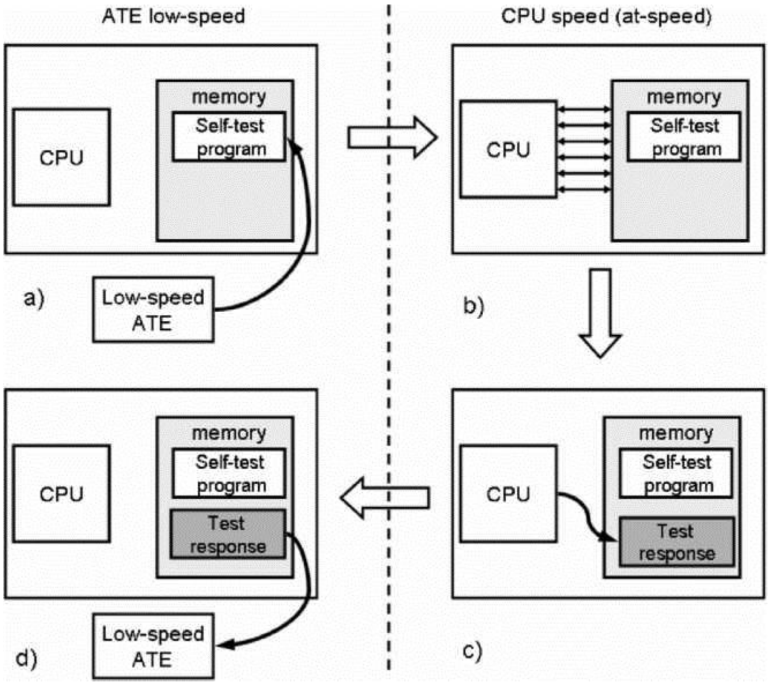


Figure 21: Software-Based Self-Test conceptual outline

SBST is a scalable, portable, and reusable methodology for high quality testing at virtually zero performance, power or circuit area overhead. SBST can be reused at different stages of the microprocessor or microprocessor-based system life cycle. SBST routines can be used during both *first silicon debug and validation* of early prototypes of a chip and *manufacturing testing* when a chip moves to full production. Besides, SBST can be used during the operation of the chip in the application field via *periodic on-line testing* for the detection of failures that did not exist or did not manifest themselves during manufacturing. In this case, SBST routines may be stored in on-chip ROM or Flash memory. On-line periodic SBST can be applied to improve reliability of low-cost systems based on embedded processors where hardware, software or time redundancy cannot be applied due to their excessive cost in terms of silicon area and/or execution time. Table 6 summarizes the different application stages of SBST and the different requirements of each stage in terms of self-test code and data size, application time and power consumption.

Table 6: Application stages of SBST and corresponding requirements

Stage	Self-Test Code/Data Stored in	Test Program Size	Test Application Time	Test Power Consumption
First silicon debug validation	Low-speed ATE On-chip cache	Large to very large	Long to very long	High
Manufacturing testing	Low-speed ATE On-chip cache	Small to medium	Short to medium	Average to high
On-line periodic testing	ROM or flash memory	Small	Short	Low

In this chapter, several different SBST strategies proposed in the research literature are briefly discussed showing the evolution of SBST and experimental data sourcing from successful applications of the SBST approach are provided wherever available.

3.1 At-speed functional testing

Traditionally, processor testing resorted in *functional testing* approaches. Functional test program development is based on either functional fault models or just the reuse of test sequences developed originally for design verification.

The latter approach has been extensively used in industry over the last two decades. Test programs generated by verification suites to verify the functionality of the processor design, are *reused* for at-speed functional manufacturing testing in an ATE-based setup. The drawback of verification-based functional testing is that it does not take account of the actual structural testability requirements of the processor, which are related to the physical defects and are formally described by fault models. Since the development of verification-based test sequences does not target structural faults (for instance single stuck-at faults) but rather processor functionality and compliance with the processor's ISA, when fault graded with respect to a structural fault model, the resulting fault coverage does not usually meet the required test quality goals. To increase the structural fault coverage, the functional-based test programs are usually augmented with manually written code by engineers with substantial knowledge of the processor architecture. Despite this additional test development effort, functional test programs cannot achieve acceptable levels of fault coverage by themselves.

In functional testing, external ATE (also called *functional testers*) is used to supply test patterns to the processor, mimicking the test program execution and the interaction between the processor and the main memory, i.e. the processor's functionality. First, simulations with a processor model are performed to capture the trace at processor I/O during the execution of test program. Afterwards, the simulation trace is translated into ATE test language and stored in the ATE memory. Finally, during test application, the ATE applies the test patterns to the processor input pins to mimic the execution of instructions while at the same time it captures the test responses at the processor output pins.

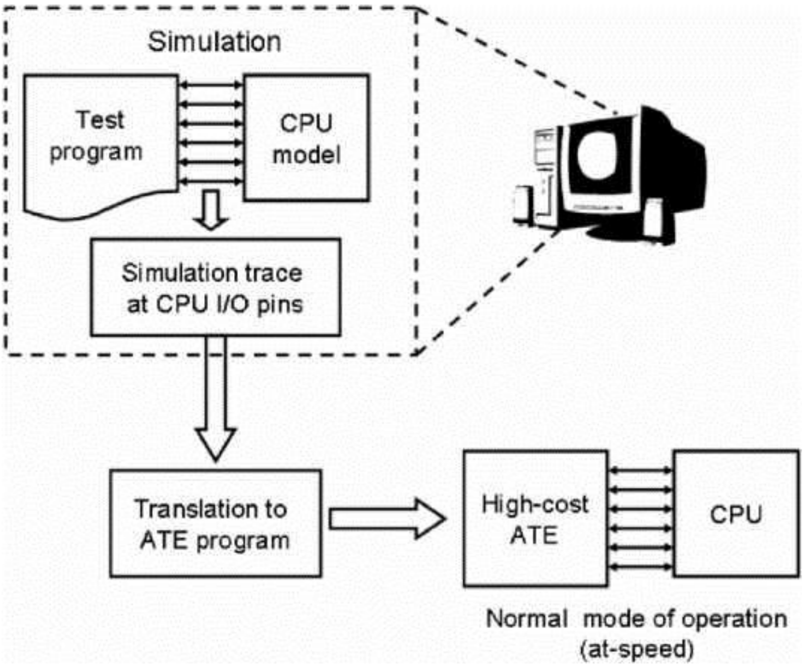

Simulation
Test program
CPU model
Simulation trace at CPU I/O pins
Translation to ATE program
High-cost ATE
CPU
Normal mode of operation (at-speed)

Figure 22: Traditional at-speed functional testing

It has already been mentioned that at-speed testing is mandatory for achieving high test quality in today's deep-submicron manufacturing technologies. Thus, the ATE used for at-speed functional testing of a processor must have the following characteristics:

- ability to supply test patterns at-speed (i.e. ATE technology needs to follow high-end microprocessors technology);
- high pin count to drive all processor I/O pins;
- large memory to store the test patterns and test responses.

However, the increasing gap between ATE frequencies and processor or SoC operating frequencies, the large test data volume, the difference between external and internal bandwidth along with the limited access to deeply embedded processor cores in complex SoCs, make external at- speed functional testing extremely costly and in many cases almost infeasible. All these drawbacks including not acceptable fault coverage as well, lead microprocessor industry to move slowly (when compared e.g. with ASIC industry) towards more intrusive, structural DFT test approaches, such as scan-based testing and BIST. However, such techniques usually have a non-trivial impact on a circuit's performance, size and power consumption and are applied with serious consideration and careful incorporation into the processor design.

The pioneer work of Thatte and Abraham [45], is considered a landmark paper in processor functional testing. Based on the register transfer (RT) level description of the processor the authors introduced a functional fault model and considered the processor as a graph model. Since then, many processor functional testing methodologies were proposed. Those approaches were either based on a functional fault model (the model of [45] or other similar ones), or based on verification principles without assuming any functional fault models at all. The functional testing work of [45] was complemented by the work of Brahme and Abraham [46] which reduces the complexity of the generated tests for the processor's instruction sequencing and execution logic. A functional model based on a reduced graph is used for the microprocessor and a classification of all faults into three functional categories is given. Tests are first developed for the registers read operations and then for all remaining processor instructions. The developed tests are proposed for execution in a self-test mode by the processor itself.

These traditional functional test approaches are characterized by the required high level of abstraction but need a large investment in manual test writing effort. Usually very little fault grading was done on structural processor netlists while high fault coverage was not guaranteed.

3.2 Software-based self-testing for microprocessors

In contrast to functional testing where an external ATE is used to drive the input test patterns and capture the output responses, SBST embeds a *"software tester*" with the form of a self-test program in the processor's on-chip memory. SBST is a *non-intrusive* approach that leverages the use of low- speed external ATE providing high quality, at-speed testing without introducing any hardware or performance overheads.

The various advantages of SBST make it a very attractive testing approach when compared to traditional at-speed functional testing or structural DFT approaches, so it comes as no surprise that numerous SBST methodologies have been proposed; a comprehensive survey can be found in [44]. The SBST approaches presented so far in the literature can be classified into three main categories. The first category includes the SBST approaches [42], [43], [47] - [49] that have a high level of abstraction and are functional in nature. A common characteristic of such SBST approaches is the almost exclusive use of randomized instructions and/or operands. The second category includes the SBST approaches [40], [41], [50] - [62] which are structural in nature and require structural fault-driven test development. The third category includes the SBST approaches [63]-[66] which combines the previous two categories such that randomized instruction test programs are followed by test programs that apply ATPG deterministic tests targeting hard-to-detect structural faults, thus constituting a “hybrid” SBST approach that provides improved fault coverage. A comprehensive list of SBST approaches from all three categories is briefly discussed in the following subsections.

3.2.1 Functional SBST approaches

The development of functional SBST programs, based on *randomized* instruction sequences and random operands has a major advantage. Due to its high level of abstraction, SBST development requires only basic knowledge of the processor architecture, and therefore requires limited test development effort and cost. However, in most cases, manual intervention is required to determine an efficient mix of instruction sequences (possibly by defining and fine-tuning instruction frequency biases) along with architecture expertise to increase fault coverage. Instruction sequences characterized as corner cases are usually targeted by specific handwritten code. Functional self-test code development does not consider any fault model and the test programs are randomly generated; thus a long test program is typically required to achieve an acceptable level of fault coverage. Despite the large number of instruction sequences, saturating behavior in fault coverage is usually observed due to pseudorandom operands used. Further increase of the random test program size is usually proved ineffective in targeting the remaining hard-to-detect faults and manual test development is a necessary supplement.

In [47], Shen and Abraham proposed a functional self-test methodology, which generates a random sequence of instructions that enumerate all the combinations of the processor operations and systematically selected operands. Test development is performed at a high level of abstraction based on ISA. However, since test development is not based on an a priori fault model, the generated tests - applicable for design validation as well - cannot achieve high fault coverage without the use of large code sequences and a considerable manual effort. When applied to the GL85 processor (model of Intel’s 8085) consisting of 6,300 gates and 244 FFs, a test program consisting of 360,000 instructions was derived and the attained single stuck-at fault coverage was

90.2%. The fault coverage was reduced to 86.7% when the responses were compressed in a signature.

In [48], Batcher and Papachristou proposed *instruction randomization self-test* (IRST) for processor cores, a pseudorandom self-testing technique. Self-test is performed with processor instructions that are randomized by a special circuit designed outside the processor core. Randomization occurs at the operand level as well. IRST does not add any performance overhead to the processor and the extra hardware is relatively small compared to the processor size (3.1% hardware overhead is reported for a 27,860 gates DLX-like RISC processor core). The obtained fault coverage after an iterative process considering different parameters for the processor core following the execution of a random instructions sequence running for 50,000 instruction cycles is 92.5%, and after the execution of 220,000 instruction cycles it is 94.8%.

In [43], Parvathala et al. proposed an automated functional self-test methodology called *functional random instruction testing at speed* (FRITS) based on the generation of random instruction sequences with pseudorandom data generated by software Linear Feedback Shift Registers (LFSR); on-chip cache is used for application. Instruction-based constraints are extracted and built into the generator to ensure generation of valid instruction sequences also ensuring that no cache misses and bus access cycles are produced during self-testing. The high-level functional nature of the proposed approach requires a large amount of cycles to be applied that makes fault grading a non-trivial task. The methodology achieved 70% fault coverage when applied to the Intel Pentium® 4 processor in an industrial environment and helped to detect the defects that escaped the normal test flow. Also, application of the approach to the integer and floating point units of Intel Itanium™ processor led to 85% single stuck-at fault coverage.

In [42], Bayraktaroglu et al. proposed the conversion of existing legacy tests, either handwritten or randomly-generated to *cache resident* tests aiming to eliminate cache misses. The basic objective of this work was to apply SBST fully avoiding the non-determinism of memory accesses in high-end microprocessors with several cache memory levels based on the use of low-cost ATE. They demonstrated their method, called *Load&Go,* to an 8-core, 32-thread Sun UltraSPARC T1 processor model.

Finally in [49], the method proposed by Corno et al. uses information feedback to improve test program quality. This approach is based on an evolutionary algorithm and can evolve small test programs and capture tar-get corner cases for design validation.

An evolutionary algorithm is a population-based optimizer that attempts to mimic Darwin's theory of evolution to iteratively refine the population of individuals (or test programs). The approach's effectiveness is demonstrated by comparing it with a pure instruction randomizer - with both systems working on an RTL description of the Leon2 processor. The proposed approach can seize three intricate corner cases that the purely random method cannot while saturating the addressed code coverage metrics. Additionally, the developed validation programs are not only more effective, but are smaller as well.

3.2.2 Structural SBST approaches

The development of SBST programs targeting structural faults using a deterministic approach clearly results in higher fault coverage when compared to functional SBST approaches where no fault model is considered at test development phase. Although SBST approaches targeting sequential fault models (such as the path delay fault model) have been presented [60] - [62], the single stuck-at fault model dominates among the SBST approaches since it reduces significantly the complexity; it is implementation technology independent while test patterns for stuck-at faults are proved effective to target most manufacturing defects. The structural SBST approaches that will be discussed in the following paragraphs, target the single stuck-at fault model.

The contribution of the work presented by Chen and Dey in [50] is twofold. First, it demonstrates the superiority of SBST for embedded processors over traditional DFT approaches such as Full Scan design and hardware Logic BIST. This is shown by applying Logic BIST to a very simple 8-bit accumulator-based processor (Parwan) and a stack-based 32-bit soft processor core that implements the Java Virtual Machine (picoJava). In both cases, Logic BIST adds more hardware overhead compared to full scan, but is not able to obtain satisfactory structural fault coverage even when a very high number of test patterns are applied. Secondly, an SBST methodology is proposed which is structural in nature, targeting specific components and fine-tuning test development to the low, gate-level details of the processor core. Initially, pseudorandom pattern sequences are developed for each processor component in an iterative method taking into consideration manually extracted constraints imposed by its instruction set. Then, test sequences are encapsulated into self-test signatures that characterize each component. Alternatively, component tests can be extracted by structural automatic test pattern generation (ATPG) and downloaded directly in embedded memory by the tester. The component self-test signatures are then expanded on-chip by a software-emulated

LFSR (test generation program) into pseudorandom test patterns, stored in embedded memory and finally applied to the component by software test application programs. Application to an accumulator-based CPU core, Parwan, consisting of 888 gates and 53 FFs, resulted in 91.4% fault coverage in 137,649 cycles using a test program of 1,129 bytes.

In [51], Chen et al. proposed a methodology that extends previous work [50] by automating the complex constraint extraction phase, while emphasizing in ATPG-based test development instead of pseudorandom. Statistical regression analysis is applied to the RT-level simulation results using manually coded instruction templates, to derive a model of the surrounding logic of the MUT. The learned model is converted into virtual constrained circuit (VCC) followed by ATPG on the VCC-MUT in an iterative way. Application of the methodology on the combinational logic in the execution stage of a processor from Tensilica (Xtensa™) with 24,962 faults resulted in 288 ATPG test patterns and 90.1% fault coverage after constrained ATPG. When the tests are applied using processor instructions in a test program of 20,373 bytes, the fault coverage for the targeted component is increased (due to collateral coverage) to 95.2% in 27,248 cycles.

In [52], Corno et al. proposed a partially automated test development approach. First, a library of macros is generated manually by experienced assembly programmers from the ISA, consisting of instruction sequences using operands as parameters. Then, a greedy search and a genetic algorithm are used to optimize the process of random macro selection among the macros set, along with selecting the most suitable macros parameters to build a test program that maximizes the attained fault coverage when the test program is applied and fine-tuned on the gate-level netlist of the processor. The approach attained 85,2% fault coverage when applied to a 8051 8-bit microcontroller design of 6,000 gates using 624 instructions.

In [53], Corno et al. proposed an automated test development approach based on evolutionary theory techniques (MicroGP), that maximizes the attained fault coverage when the evolved test program is applied to the gate-level netlist of the processor. It utilizes a directed acyclic graph for representing the syntactical flow of an assembly program and an instruction library for describing the assembly syntax of the processor ISA. Manual effort is required for the enumeration of all available instructions and their possible operands. Experiments on a 8051 8-bit microcontroller design of 12,000 gates, resulted in 90% fault coverage.

In [54], Kambe et al. proposed a template generation methodology for hierarchical test generation targeting structural faults. According to the methodology, gate-level test generation is performed for each MUT, and a test program is generated to justify test patterns from primary input (PI) to the MUT and propagates test responses at instruction level. The proposed methodology enumerates possible templates considering dependence of instructions each of which involves one or more data transfers between registers. In order to justify value of MUT inputs, a concept of adjacent registers of the MUT is introduced that makes it possible to consider input spaces of the MUT determined by signals from other modules as well as signals directly from registers. Templates are generated considering dependence of instructions each of which invokes one or more data transfers between registers. The approach is demonstrated on an accumulator-based 8-bit CPU core, Parwan. Out of 276 templates generated for testing the ALU of Parwan, 12 templates contributed to the fault coverage, and the fault coverage achieved for the ALU was 99.44%.

In [55], Kranitis et al. introduced a high-level structural SBST methodology, showing for the first time that small deterministic test sets, applied by compact test routines provide significant improvement when applied to the same simple accumulator-based processor design, Parwan, which was used in [50]. Compared to [50], the methodology described in [55] requires 20% smaller test program using 923 bytes, 75% smaller test data and almost 90% smaller test application time using 16,667 cycles. Both methodologies achieve single stuck-at fault coverage slightly higher than 91% for the simple accumulator-based Parwan processor.

Despite the successful first application of the approach of [55], scaling from simple accumulator-based processor architectures to more realistic ones in terms of complexity like contemporary complex processors implementing commercially successful ISAs (i.e. RISC), brings out several test challenges that remained unsolved. These challenges arise when high-level test development is applied to complex processor architectures that contain large functional components (i.e. fast parallel multipliers, barrel shifters, etc.) and large register banks, while trying to keep the test-cost as low as possible. In [40], Kranitis et al. addressed low-cost SBST challenges by defining different test priorities for processor components, showing that high- level self-test code development based on ISA and RT-level description of a processor can lead to low test cost without sacrificing fault coverage independently of the gate-level implementation. The methodology was applied to two processors: Plasma/MIPS with simple 3-stage pipeline and MIPS R3000 application specific instruction set processor (ASIP) with 5-stage pipeline designed using the ASIP/Meister design environment.

In [41], Paschalis and Gizopoulos identified the stringent characteristics of an SBST test program to be suitable for on-line periodic testing of embedded processors. SBST for on-line periodic testing can be applied to improve reliability of low-cost embedded systems based on embedded processors where hardware, software or time redundancy cannot be applied due to their excessive cost in terms of silicon area and execution time. A new classification and test priority scheme more fine-grained than in [40] was proposed. Both types of permanent and intermittent faults are detected by a small embedded test program with test execution time much less than a quantum time cycle.

In [56], Sanchez et al. proposed an automatic methodology to transform a test set originally developed for manufacturing test in a test set suitable for on-line testing. The generated programs are suitable for non-concurrent periodic on-line testing as well as for shutdown or startup testing. While the new test set is likely to contain a larger number of programs, these programs are shorter and completely independent (i.e. they can be executed at different times and do not rely each on the results of the previous ones), and thus perfectly fit a non-concurrent on-line test scheme. The transformation of the test set is performed in two phases: first the original programs are simulated with a special instruction-set simulator that for each instruction generates a spore, i.e. a small program able to fully replicate the processor behavior. Second, an evolutionary algorithm is used to collapse the set of spores into a test set. The proposed approach is able to guarantee the same fault coverage on all functional units. Experimental results were provided targeting the ALU and Control Units of an 8-bit 8051 processor core. The initial test set is compact in size but requires a long time to be executed and is usually designed to be run without regarding sharing constraints. The final on-line test set is larger in size, but composed of small and extremely fast programs that can be freely scheduled. Both test sets guarantee the same fault coverage on the target units.

In [57] and [58], Gizopoulos et al. identified testability hotspots in processor pipeline logic and proposed a generic SBST methodology that enhances existing SBST programs [40], to target more effectively the pipeline logic of more sophisticated pipelined processors. They first identify the testability hotspots of the pipelining logic, applying existing SBST programs (generated according to the methodology by Kranitis et al. and targeting the processor's functional components) to two fully pipelined RISC processor models: miniMIPS and OpenRISC 1200. The proposed automated methodology complements any other SBST generation approach that targets functional components. It analyzes the data dependencies of an existing SBST program and considers the basic parameters of the pipelined architecture (number of stages, forwarding paths, etc.) and the memory hierarchy system (virtual- and physical-memory regions) to generate an enhanced SBST program that comprehensively tests the pipelined logic. The experimental results on the two processors show that the enhanced SBST program achieves significant fault coverage improvements for the pipelined logic (19% improvement on average) and for the total processors (12% improvement on average).

The contribution of the work presented by Kranitis et al. in [59] is twofold. First, a reliability analysis and a cost function was introduced in order to minimize the test cost incurred when selecting a periodic SBST strategy, and achieve high detection probability. Reliability analysis was based on a two-state Markov model for the probabilistic modelling of intermittent faults for optimal periodic testing is introduced. Then, an SBST strategy for on-line SBST of pipelined embedded processors was proposed that enhances SBST programs for manufacturing [40] and on-line testing [41]. The proposed strategy was demonstrated by applying it to a 5-stage fully pipelined RISC embedded processor, Athena. Experimental results provided showed 8.2% fault coverage improvement for the entire processor and fault coverage improvements of 26% for the pipeline logic.

3.2.3 Hybrid SBST approaches

A common characteristic among these “hybrid” SBST approaches [63] - [66] is that randomized instruction test programs are followed by test programs that apply ATPG deterministic tests targeting hard-to-detect structural faults.

In [65], Wen et al. introduced an SBST methodology that employs random test program generation (RTPG) as a baseline with deterministic target test program generation (TTPG) as a supplement, in order to provide tests specifically targeting faults that are hard-to-test for RTPG. The proposed TTPG method utilizes simulation results to develop learned models for the surrounding modules of the block under test. Simulation-based TTPG is performed similar to previous works; however, arithmetic and Boolean learning techniques are used instead of statistical regression to develop learned models for the surrounding logic of the MUT. These techniques offer the advantage of being deterministic in nature, in contrast to regression that is a statistical method. Additionally, Boolean learning can also handle logic-intensive modules in which regression is not effective. Then, the learned models replace the surrounding modules around the block in the actual test generation process. Because the learned models are much simpler to handle, this method minimizes the cost of functional TPG. The methodology is applied to the controller and ALU of the OpenRISC 1200 processor. When RTPG is applied in the “controller” module fault coverage saturates around 62.14%, while on the other side, TTPG generates 134 valid test patterns and detects 4967 faults including all faults that RTPG can detect, for an overall fault coverage of 69.39%. For the ALU module, after application of 100K RTPG test patterns, TTPG is applied and the combined fault coverage is 94.94%.

In [63], Gurumurthy et al. introduced a novel technique to map precomputed test patterns, generated by commercial ATPG tools, into sequences of instructions, based on the ISA of the processor under test. The technique applies at the RT-level source code of the processor, at the module level. It uses bounded model checking in order to produce automatically a counterexample which will contain an instruction sequence that generates the pre-computed test pattern. First, a bound is defined for the bounded model checker (BMC) for each step of the process, taking into consideration the pipeline depth, the stall/reset mechanism of the processor, the forwarding mechanism and the number of cycles of the instructions. Then, every test pattern for each module should be manually transformed into linear temporal logic (LTL) property. LTL property is negated and passed to the BMC. Additionally, the instruction set of the processor is passed to the BMC in order to constrain its input space. BMC checks partial correctness of the property and generates a counter-example in case the property fails within the bound. If a counter-example is not generated, pre-computed test patterns are characterized as functionally infeasible, otherwise processor instruction sequence containing test pattern is included in counter-example. The technique is applied to both controllability and observability stages and results to a combination LTL property of both stages. Although controllability is fully controlled in this technique, in observability stage spurious counter-examples can be generated that do not ensure propagation of outputs to observable points, thus they have to be refined entirely manually. Experiments were performed on OpenRISC 1200. Initially, a random test program of 36,750 instructions was generated in order to fault grade the processor. The fault coverage saturated around 68% and the remaining hard-to-detect fault list was split based on modules and passed through a commercial ATPG tool in order to obtain the pre-computed patterns. Those pre-computed test patterns were applied to the presented technique. In a total of 22,633 test sequences, 6,765 were identified to be functionally infeasible uncontrollable sequences. On the remaining, sequences of instructions were generated for some of the patterns in ALU, Control and Operandmuxes modules of OpenRISC 1200 and example instruction sequences were given.

In [64], Gurumurthy et al. proposed a new technique that fully automates the process of functional test generation targeting specific faults. The technique supplements the observability part of the automated mapping technique of pre-computed test patterns, generated by commercial ATPG tools, into sequences of instructions proposed in [63] that required manual effort for the propagation of test responses. The proposed technique applies at the RT-level source code of the processor in module-level.

It uses Boolean difference, LTL and bounded model checking (BMC) in order to map module-level test responses into instruction sequences. Experiments were performed on OpenRISC 1200 processor as in [63]. Again, in order to focus on hard-to-detect faults, a random test program of 36,750 instructions was generated and the processor was fault-graded. The fault coverage saturated around 68% and the hard-to-detect fault list formed the base list of the proposed technique. The base list was sorted based on module-level and the overall technique was used for every module. Even though the mapping efficiency of most of the modules is above 90%, the overall mapping efficiency was 71% due to low efficiency of ALU and LSU modules. In a total of 17,319 test sequences, 9,708 were found to be not mappable within the bound, thus no counter-example was produced and were rejected. The remaining test sequences were successfully mapped and increased the fault coverage of the processor to 82%.

Finally in [66], Kranitis et al. proposed a hybrid-SBST methodology for efficient testing of commercial processor cores that effectively uses the advantages of various SBST methodologies. Self-test programs based on deterministic structural SBST methodologies (using high-level test development and gate-level-constrained ATPG test development) combined with verification-based self-test code development and directed RTPG constitute a very effective H-SBST test strategy. The proposed methodology applies directed RTPG as a supplement to improve overall fault coverage results after component-based self-test code development has been performed. An advantage of this strategy is that it avoids the use of large RTPG programs that result in an excessive number of cycles and prohibitive test application time during manufacturing test. A test program following these principles has been developed and applied to a commercial, fully pipelined benchmark, OpenRISC 1200. Experimental results showing test coverage of more than 92% demonstrate the effectiveness of the proposed methodology.

3.2.4 Recent trends in SBST approaches

The most recent architectural advances dictate the integration of multiple cores and hardware threads on a single die to deliver high computing power by exploiting thread-level execution parallelism. Following these advances, a recent trend in SBST research is to scale SBST techniques in multiprocessor and multi-threading architectures.

Two recent approaches aim to reduce the test application time by exploiting core or thread level parallelism. In [67], Apostolakis et al. have proposed porting an SBST approach from the unicore case to a bus-based symmetric multiprocessor (SMP) architecture. Their self-test scheduling algorithm at the core level significantly minimizes the time overheads caused by data cache coherency invalidations and intense bus contention among cores. They've demonstrated their methodology in two, four, and eight-core versions of a multiprocessor based on an Open-RISC 1200 core. Experimental results show that the methodology achieves more than 91% total stuck-at fault coverage for all multicores, while reducing test application time by more than 24% compared to the fastest alternative.

In [68], Foutris et al. have presented a multithreaded software-based self-test (MT-SBST) methodology that targets both the optimization of test execution time and the improvement of the fault coverage of the thread-specific control logic. The proposed methodology is based on a multithread scheduling algorithm that achieves a very efficient balance between self-test program execution time and fault coverage of the thread-specific control logic and is solely based on easy-to-obtain runtime statistics of the single-threaded execution of the self-test programs. The methodology has been applied to the chip-multithreading architecture of Sun's OpenSPARC T1, which integrates eight CPU cores, each supporting four hardware threads. The experimental results have shown that the proposed multithread scheduling algorithm significantly speeds up the execution time of test program at both the core-level (up to 3.6X) and the processor-level (up to 6.0X) compared to the single-threaded execution.

3.3 Software-based Self-Test for embedded cache testing

In the case of on-line memory testing, SBST has increased flexibility to apply any kind of March tests. Hence, SBST can face successfully the challenges of on-line testing of small memory arrays that partially or totally lack programmable MBIST.

Research community has acknowledged the potential of utilizing SBST techniques in cache testing and several SBST approaches have been proposed [69] - [79].

In [69], the first systematic approach for transforming March B test algorithm for in-system testing of Intel 860 processor cache is proposed, however instruction cache testing is only outlined.

In [70], Sosnowski proposed a generalized and uniform March test transformation methodology of designing user test programs for data and instruction caches of various organizations. It covers instruction and data caches with different write and replacement policies, multilevel and distributed caches. The methodology is proposed for in-system testing of cache memories with various organizations by taking advantage of features such as enable/disable or freeze, programmable cachability of memory pages, deterministic line replacement algorithms (or selective enabling of cache memory banks), etc. However, the proposed methodology has not been demonstrated to any real processor benchmark.

In [71], Al-Harbi et al. proposed a methodology to transform March tests and use it to obtain new versions of March B and March X tests for in-system testing of faults in the tag parts of direct mapped caches. The key aspect of the proposed transformation is that it leads to a test with a significantly lower complexity while preserving the fault detection capability of the original test without necessarily recreating an identical sequence of reads and writes. Furthermore, the resulting cache tests do not require the hardware modification needed by the previously proposed cache test. The methodology targets only the data cache memory tag without providing implementation details.

In [72], the methodology that is presented in [70] is enhanced to exploit microprocessor’s performance monitoring hardware and on-line hardware detectors to improve test observability. However, the proposed enhancements are only outlined and the methodology has not been applied to any benchmark.

In [73], Tuna et al. proposed an SBST approach to develop self-test programs relying on March tests for the data array of both instruction and data caches. Experimental results for traditional memory faults are provided for a MIPS R3000 processor model using several March tests.

In [74], Alpe et al. proposed a methodology to translate generic March tests into equivalent versions for in-system testing of both directory and data array of set-associative caches with write-back or write-through policy. Among the different types of replacement algorithms for set-associative caches the methodology focus on memories implementing the Least Recently Used (LRU) replacement. The main goal is to propose a translation methodology providing tests that preserve both the original fault coverage and (wherever possible) the complexity of the original March test.

In [75], Lin et al. proposed a software-based methodology for testing memory arrays and logic modules of a direct-mapped data cache. The proposed methodology is applied to a direct-mapped data cache embedded in a Linux-verified ARM-compatible processor and results in 100% fault coverage for six conventional RAM fault models and 98.03% fault coverage or 99.13% test efficiency for the collapsed single stuck-at faults of the logic modules.

In [76], Perez et al. proposed a hybrid SBST approach to test data and instruction cache controllers by combining instruction-based pattern generation and an I-IP module insertion for observability. Experimental results for the cache controllers of OpenRISC 1200 processor are provided.

In [77], Perez et al. presented an algorithmic-based strategy to test the replacement logic in set-associative caches that implement a deterministic replacement policy. The methodology is suitable for post-production testing, for incoming inspection, and for the on-line testing of both stand-alone processors and processor cores embedded in SoCs. The proposed algorithm can be tailored to different cache configurations, both in terms of cache size and organization, and in terms of writing strategies (i.e., write-back and write-through). The methodology is based on modeling the behavior of the replacement mechanism of the cache as an FSM machine. To experimentally evaluate the real performance of the proposed approach, a simulation scheme has been implemented in assembly code and fault simulations results have been gathered for a pipelined processor whose cache controller implements the LRU approach.

In [78], van de Goor et al. presented the capabilities and limitations of CPU-based at-speed memory testing based on test routine examples for an ATMEL RISC microcontroller. Such SBST routines can be also adopted for testing cache arrays.

Recently in [79], Di Carlo et al. proposed a methodology to exploit the ISA of a processor to translate generic March tests into SBST programs for set-associative cache memories. The proposed methodology concentrated on testing instruction caches. The methodology applies state-of-the-art memory test algorithms to embedded cache memories without introducing any hardware or performance over-heads. The proposed methodology has been demonstrated on a test program for the instruction cache of the LEON3 microprocessor.

4. SBST METHODOLOGY FOR CACHES

4.1 Cache arrays testability challenges

In this section, the testability challenges that occur when SBST approaches are utilized to apply March test to the cache memory arrays, are introduced. But first, an overview of the main characteristics of all the available cache organization schemes is presented.

4.1.1 Cache organizations overview

Cache stores chunks of data (called cache blocks or cache lines) that come from the backing store. Because a cache is typically much smaller than the backing store, there is a good possibility that any particular requested datum is not in the cache. Therefore, some mechanisms must indicate whether any particular datum is present in the cache or not. C*ache tags* fill this purpose [80]. Tags are valid address parts and comprise a list of valid entries in the cache, with one tag per data line.

Cache typically divides its storage into sets and assign blocks to sets according to every block ID, which typically is a specific part of the block address. There are three basic cache organizations: *direct mapped*, *fully associative* and *set associative*. A direct mapped cache has sets with only one block in its set, a fully associative cache has only one set that encompasses all blocks in the cache, thus a Least-Recently Used (LRU) algorithm is usually used to place data into cache and a set associative cache has more than one set and each set in the cache incorporates more than one block that are placed by LRU inside a set. In processor design, data and instruction L1 caches are usually organized as direct mapped and set-associative caches whereas data and instruction TLBs are always organized as fully associative caches.

A typical L1 cache organization comprises of at least two SRAM memory arrays (or two SRAM arrays per set in set-associative organizations) - the data array and the tag array - whereas a fully-associative TLB organization comprises of one SRAM array - the data array - and one CAM array - the tag array. CAM is a special type of memory that compares all the stored data in parallel with incoming data and is utilized in the tag part of fully associative caches to speed up the tag comparison.

Figure 23 shows typical cache block diagrams for all three cache organizations: direct mapped organization (a), set associative organization (b) and fully associative organization (c).

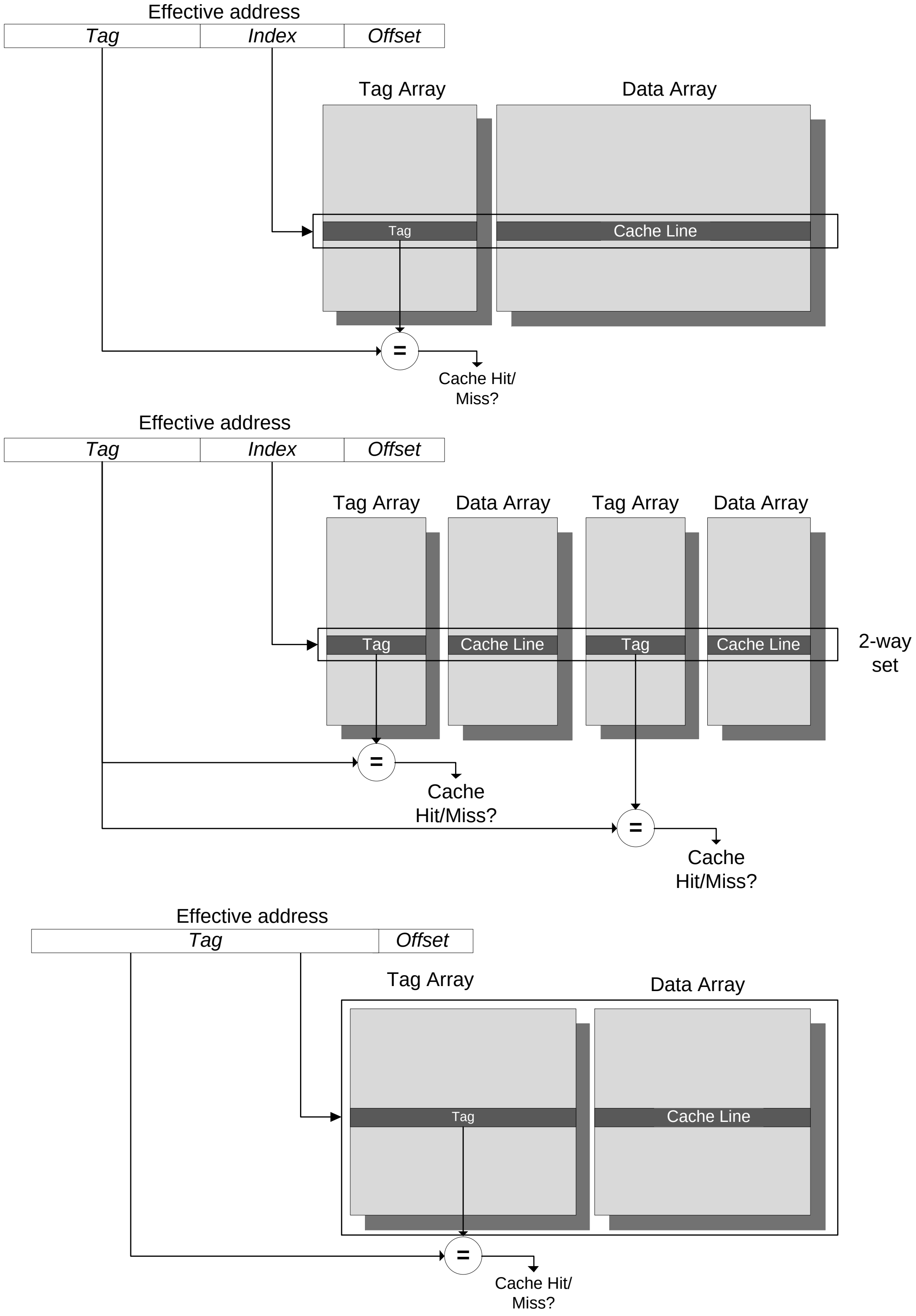


Figure 23: Cache organizations: a) Direct mapped b) Set-associative c) Fully associative

Index is the part of address that provides either the exact cache line location in direct mapped caches or the set number that the cache line will be placed in set associative caches, *offset* is the address part that access blocks (usually words) inside cache line and the rest high part of address is stored as *tag*. Tag size is counter proportional to cache size. In direct mapped cache each memory block can be stored in a unique cache line according to the index value. In a *k-way* set associative cache topology each block can be located in one of the *k* lines of a set that is defined by index. Inside the set, a cache line is selected usually by the LRU replacement algorithm. In a fully associative cache, a memory block can be stored in any cache line based on the LRU replacement algorithm. To conclude, cache memories can be divided into two more categories, based on their write policy:

- *Write-through* policy: The information is written both to the block in the cache and to the block in the lower level of memory, providing simplicity and data coherency.
- *Write-back* policy: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced thus reducing memory traffic and increasing performance

4.1.2 Cache arrays testability challenges

In this section, the testability challenges for the cache arrays that have been described above will be presented. Further down, those cache arrays will be denoted as DL1-Data, DL1-Tag, IL1-Data and IL1-Tag for the data and instruction L1 cache whereas for TLBs those arrays will be denoted as DTLB-Data, DTLB-Tag, ITLB-Data and ITLB-Tag for the data and instruction TLB, respectively.

Table 7: Cache arrays testability challenges

Testability Challenges	***Cache arrays***							
	DL1 Data	***DL1 Tag***	***IL1 Data***	***IL1 Tag***	***DTLB Data***	***DTLB Tag***	***ITLB Data***	***ITLB Tag***
Direct access from generic ISA	×	×	×	×	×	×	×	×
Indirect March write (controllability)	✓	✓	×	✓	×	×	×	×
Indirect March read (observability)	✓	×	×	×	×	×	×	×
Data Backgrounds (DBs) composition	✓	×	×	×	×	×	×	×
Ascending Address Order	✓	✓	✓	✓	×	×	×	×
Descending Address Order	✓	✓	×	×	×	×	×	×
March Compare operation (Tags)	-	-	-	-	-	×	-	×

All these cache arrays (either for L1 caches or TLBs) are implicitly accessed because they are not directly visible to the assembly programmer through the ISA. Hence, applying test patterns and observing the test responses through a software test routine is challenging. The challenges of accessing and thus testing those implicitly accessed cache arrays are summarized in Table 7.

DL1-Data array can be easily accessed indirectly by using load/store generic instructions. Data backgrounds (DBs) required by March tests can be easily composed by initializing the main memory with the corresponding test vectors by using store instructions and then by writing them to the DL1-Data array (March write operation) either by a store instruction that misses (when write-allocate is supported) or by a load instruction that misses and refills the cache lines. Generic load instructions can afterwards read the test vectors to registers, hence March read operations can be mapped to such instructions. Indirect access for implementing both March writes and March reads to the rest of the cache arrays is challenging.

DBs can be written to DL1-Tag array by using a store/load instruction that misses and refills a cache line, while DBs can be written to both IL1-Data and IL1-Tag array by using a call instruction and the miss & refill mechanism of instruction cache. Indirect March writes to all four DTLB arrays can be implemented by utilizing the miss & refill mechanism of TLB through well-designed TLB misses for March writes. However, applying March write operations in fully associative TLB arrays by utilizing LRU replacement policy can be challenging in certain March test implementations (e.g. when a write after read of the same DB is required in a memory cell) due to the lack of a mechanism for selecting distinct TLB entries multiple times.

Contents of DL1-Tag, IL1-Data and IL1-Tag arrays and all four TLB arrays are not directly readable by generic ISA instructions and thus mapping March read operations to an ISA is challenging and can be only implicitly realized by detecting unexpected L1 cache or TLB misses. March reads to all these arrays can be implemented by utilizing the miss & refill mechanism of both L1 caches and TLBs through well-designed cache hits, since March reads always validate successful March write operations. Since, DL1-Tag, IL1-Tag and all four TLB arrays store address information a successful March read in these arrays will lead to a cache hit whereas an unsuccessful March read should cause a cache miss. Those unpredicted cache misses should be monitored in order to identify a faulty array. IL1-Data array stores actual instructions. In this case, an unsuccessful March read will lead to unpredicted system behavior since a faulty instruction will be fetched.

Every time a cache line is inserted into the DL1-Data or the IL1-Data array, a tag entry is also introduced. Furthermore, every time a virtual address is generated either by a store/load instruction (or an instruction fetch), its virtual page number is stored to the DTLB-Tag (or ITLB-Tag) and its physical page number is issued to the DTLB-Data (or

ITLB-Data) array. As all these arrays store address information and page numbers, DBs should be valid addresses and valid page numbers, respectively. Hence, DBs composition is challenging for both L1 tag arrays and all four TLB arrays due to potential limitations in accessing several memory segments. This testability challenge is further exacerbated by the fact that in most L1 cache and TLB organizations a set of control bits is also included in these arrays. These extra control bits have limitations when accessed indirectly through an ISA; these limitations should be considered when implementing March operations through an SBST routine. IL1-Data array encounters serious limitations in March test DBs composition since DBs must be composed by valid instructions. Hence, the limitations mainly due to opcode encoding have to be encountered.

Implementation of ascending/descending address order as required in all March tests is also challenging in most of the cache arrays. While the implementation of an ascending address order is straightforward for L1 instruction cache arrays (IL1-Tag and IL1-Data), the implementation of a descending address order is challenging, because instructions are normally fetched in ascending address order during system operation. Moreover, for all four TLB arrays (DTLB-Data, DTLB-Tag, ITLB-Data and ITLB-Tag), the implementation of both ascending and descending address order is challenging due to the fully associative organization and the LRU replacement policy.

Finally, TLB tag arrays (both DTLB-Tag and ITLB-Tag) are implemented with a CAM memory and should be tested for both storage and comparison faults. In order to implement a March test for comparison faults [17], an extra March compare operation should be indirectly implemented through ISA. Such an operation cannot be implemented through the native miss & refill mechanism. In order to implement a March compare operation, a miss-no-refill mechanism is required which is not included in most processors.

The methodology that is introduced in this thesis overcomes all these testability challenges for all cache arrays of both L1 caches and TLBs and optimizes the SBST routines in terms of test time and test code size by exploiting DCA instructions.

4.2 DCA instructions in modern ISAs

So far, previous SBST approaches cannot successfully overcome all the above mentioned challenges by using generic instructions to access cache arrays both for write and read operations. Fortunately, modern ISAs include dedicated instructions for debug-diagnostic and performance purposes that provide direct controllability and observability of cache arrays. These instructions are extremely suitable for cache and TLB SBST; we use the term Direct Cache Access (DCA) instructions to refer to them.

In order to gain direct access to the cache arrays for all three cache organizations (direct mapped, set-associative and fully-associative) and implement a software-based March test, an ideal DCA instruction needs to contain the following fields:

Fields for selecting cache/TLB content:

- Way Selection (WS) field.
- Set Selection (SS) field.
- Line Word Selection (LWS) field.

Field for selecting internal cache array:

- Data/Tag Array Selection (AS) field.

Field for selecting March operation:

- Write/Read/Compare operation (WRC) field

Field for addressing register/memory for fetching DBs:

- From/To data Address (A) field

An ideal DCA instruction that contains all these fields gains direct access to any cache array, hence it can apply March operations through ISA to these arrays in a very effective way and overcome all the above described testability challenges. DCAs that access direct mapped caches should contain only fields *SS* and *LWS* while DCAs for accessing set associative L1 caches should contain all three *WS*, *SS* and *LWS* fields. DCAs that access fully associative caches and TLBs should contain only *WS* and *LWS* fields (fully associative caches contain 1 set with many ways) respectively. When the cache organization imposes a uniform cache line (e.g. TLBs) *LWS* field is not required. Finally, if the cache organization does not comprise a CAM memory (e.g. L1 caches), the March operation selection field can be renamed to *WR* field (only write/read operations, no compare).

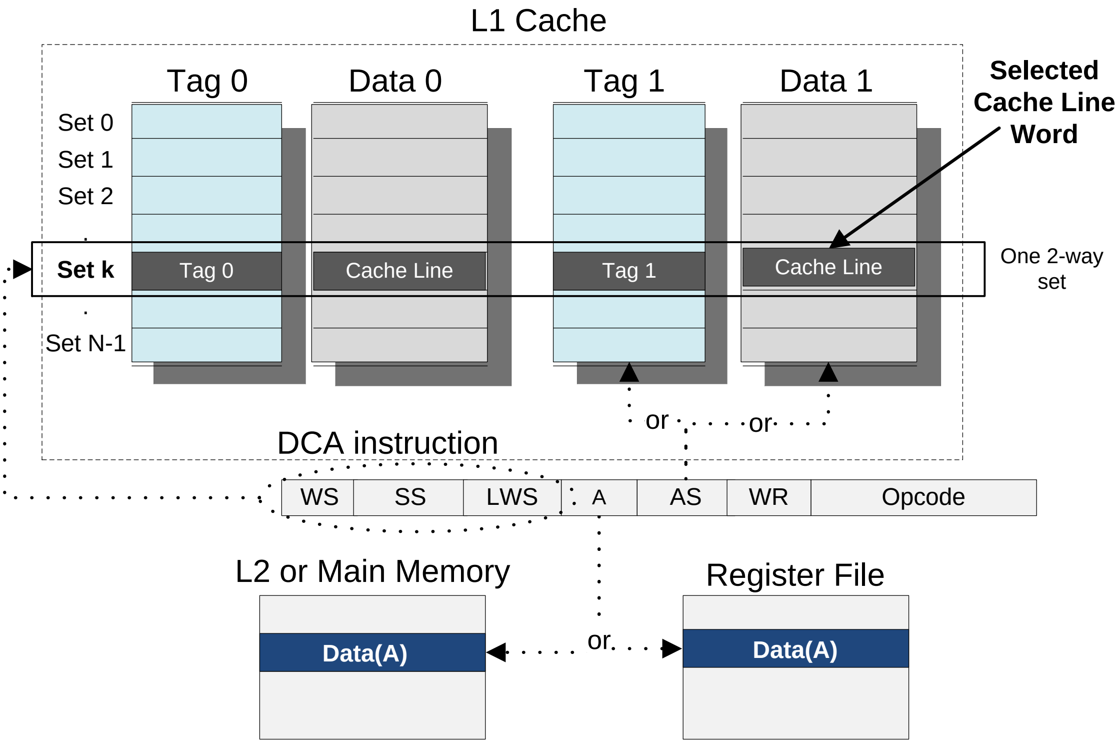


Figure 24: Ideal DCA instruction for 2-way set associative L1 cache

In Figure 24 and Figure 25, ideal DCA instructions and the way that every field is utilized to access a 2-way set associative L1 cache and a TLB, are presented, respectively.

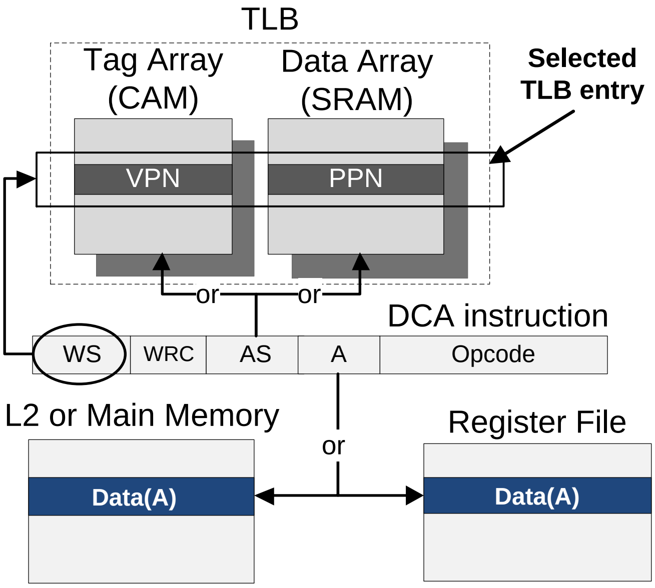


Figure 25: Ideal DCA instruction for TLB

In detail, in L1 set-associative caches the data or tag array (selected by the *AS* field) is accessed both for write and read operation (selected by the *WR* field). The selection of a word inside a cache line is controlled in three steps. First, the *SS* field selects the set, then, the *WS* field selects the cache way and finally the *LWS* field selects the word inside the cache line. Furthermore, all DBs can be composed by initializing either a general purpose register or a memory location that can be accessed by the *A* field. In TLBs, the data or tag array of the TLBs can be accessed by controlling the *AS* field. The March operation can be selected with the *WRC* field to write, read or compare a selected TLB entry. Compare operation is valid only for tag array. TLBs are fully associative arrays; hence the *WS* field is needed in the ideal DCA instruction in order to gain direct access to every TLB entry. Note that, such an instruction has no limitation in accessing a cache either in ascending or in descending order.

In practice, such an ideal DCA instruction does not exist in ISAs but it can be indirectly implemented by combining a set of existing DCA instructions that together totally cover all fields of the ideal instruction. Representative examples of such special purpose instructions, which can be considered as DCAs, are present in RISC architectures, such as MIPS, ARM and SPARC architectures [81]-[83] and in CISC architectures such as x86 architectures [84] .

MIPS architectures implement special instructions for debug-diagnostic purposes that can directly access both L1 cache and TLB data and tag arrays (*CACHE* instruction and *TLBWI*, *TLBR* and *TLBP* instructions). These instructions are part of the system coprocessor CP0 instructions that are implemented in modern MIPS architectures (e.g. MIPS R10000) and can write and read L1 cache and D-TLB contents with the content of either a general purpose register (*CACHE* instruction) or the ReadHi or ReadLo registers (*TLBWI* and *TLBR* instructions), respectively without executing store/load instructions. Moreover, *TLBP* instruction implements a compare operation for the TLB. Hence, these instructions have similar fields with the ideal DCA instruction and can be considered as DCAs that can effectively implement March operations.

ARM architecture implements system control coprocessor (CP15) debug operations (*MRC* and *MCR* instructions) for accessing both the L1 cache and TLB arrays (by utilizing register 15) for directly write, read (and compare for TLBs) the cache content. These instructions are executed in secure privileged mode and provide great visibility of the cache arrays by interrupting the program flow to execute them. They transfer the array contents from/to system array debug data registers without executing store/load instructions. Therefore, these instructions can be also considered as DCA instructions.

SPARC ISA implements alternate space identifier (ASI) store/load instructions (e.g. *sta/lda* instructions in LEON3 and *stxa/ldxa* instructions in UltraSPARC T1). These instructions are utilized to access embedded RAMs through a SPARC diagnostic access bus that bridges these embedded RAMs with the main memory or the processor’s register file. Every embedded RAM, including L1 cache arrays and TLB arrays, can be accessed by using ASI store/load instructions in order to implement write/read operations. Especially, for L1 cache arrays, dedicated ASIs for all four cache arrays (DL1-Data, DL1-Tag, IL1-Data and IL1-Tag) are defined to access cache in hypervisor level. For TLB arrays, dedicated ASIs for all four TLB arrays are also defined to access TLBs in hypervisor level. This way, the dedicated ASI field for each cache array of the ASI store/load instructions maps to the *AS* field of the ideal DCA instruction. Also, the ASI store/load instructions contain fields that map to *WS*, *SS* and *LWS* fields of the ideal DCA instruction, respectively. Hence, these instructions, when combined, cover all fields of the ideal DCA instruction and can be effectively used to implement March operations.

CISC x86 architectures implement model specific registers (MSRs) that are used to provide access to features that are tied to implementation aspects of x86 processors. One of these features is to gain test access to physical structures such as L1 caches, TLBs and branch target buffers. In Intel’s ISA, *WRMSR* and *RDMSR* instructions are implemented to access these MSR’s and activate these model specific additional features. Therefore, these instructions (*WRMSR* and *RDMSR*) when exploited to access cache arrays can be also considered as DCA instructions in order to implement March operations.

Finally, instructions that implement the L1 cache prefetch mechanism can be considered as DCA write-only instructions to achieve cache controllability. Prefetch mechanisms are utilized to initialize cache contents in order to reduce cache miss ratio of applications. Such prefetch mechanisms are present in all modern microprocessors (e.g. *dcbt* instruction in IBM’s PowerPC, *prefetch0* in Intel’s Pentium, *pld* instruction in ARM’s Cortex and Intel’s Xscale). The instructions that implement these mechanisms initialize cache sets with main memory blocks that are selected by an instruction field. The prefetch instructions, when utilized, can access any L1 cache line, but not any word inside the cache line for write operations. Prefetch instructions have to be combined with generic instructions for cache observability.

4.3 March test notations

A March test consists of a sequence of March elements. Each March element consists of a sequence of operations (writes and reads). The way the sequence proceeds to the next step is determined by the addressing order and is denoted by the “↑” and “↓” symbols for ascending or descending order, respectively. March operations are symbolized as W_0, W_1, R_0, R_1 for write and read operations. In word-oriented memories, March write and read operations are applied to several runs for different values that are called data backgrounds (DBs) [85] and are symbolized as $W_{DB}, W_{\overline{DB}}, R_{DB}, R_{\overline{DB}}$, respectively. In case of March tests for CAM comparison faults, an extra March operation is needed, the compare operation and is symbolized as $C_{DB}, C_{\overline{DB}}$. A word-oriented March test that targets storage faults - March SS [11] - and a March test that targets comparison faults - March NCDA [24] -, is shown below:

$$\text{MarchSS}: \left\{ \begin{array}{l} \underbrace{\uparrow (W_{DB})}_{M_0}, \underbrace{\uparrow (R_{DB}, R_{DB}, W_{DB}, R_{DB}, W_{\overline{DB}})}_{M_1}, \underbrace{\uparrow (R_{\overline{DB}}, R_{\overline{DB}}, W_{\overline{DB}}, R_{\overline{DB}}, W_{DB})}_{M_2}, \\ \underbrace{\downarrow (R_{DB}, R_{DB}, W_{DB}, R_{DB}, W_{\overline{DB}})}_{M_3}, \underbrace{\downarrow (R_{\overline{DB}}, R_{\overline{DB}}, W_{\overline{DB}}, R_{\overline{DB}}, W_{DB})}_{M_4}, \underbrace{\downarrow (R_{DB})}_{M_5} \end{array} \right\}$$

$$\text{March NCDA}: \left\{ \begin{array}{l} \underbrace{\uparrow (W_{DB})}_{M_0}, \underbrace{\uparrow (R_{DB}, W_{\overline{DB}})}_{M_1}, \underbrace{\uparrow (C_{walk0})}_{M_2}, \underbrace{\uparrow (R_{\overline{DB}}, W_{DB})}_{M_3}, \underbrace{\uparrow (C_{walk1})}_{M_4}, \\ \underbrace{\downarrow (R_{DB}, W_{\overline{DB}}, C_{\overline{DB}})}_{M_5}, \underbrace{\downarrow (R_{\overline{DB}}, W_{DB}, C_{DB})}_{M_6}, \underbrace{\downarrow (R_{DB})}_{M_7} \end{array} \right\}$$

4.4 SBST March test development

The methodology targets all cache arrays for both data and instructions (either L1 caches or TLBs) and is suitable for all three cache organizations with any write policy. An SBST technique that targets L1 caches cannot be cache resident, since the actual L1 cache is under test. However, this is not a limitation in case of on-line testing, since the test routines can be stored and executed from either L2 cache or the chip’s main memory that is available at test time. Moreover, when the SBST methodology targets TLBs, the SBST routine should be placed in a non-pageable memory location that is not cached to the instruction TLB since the actual TLB arrays are under test.

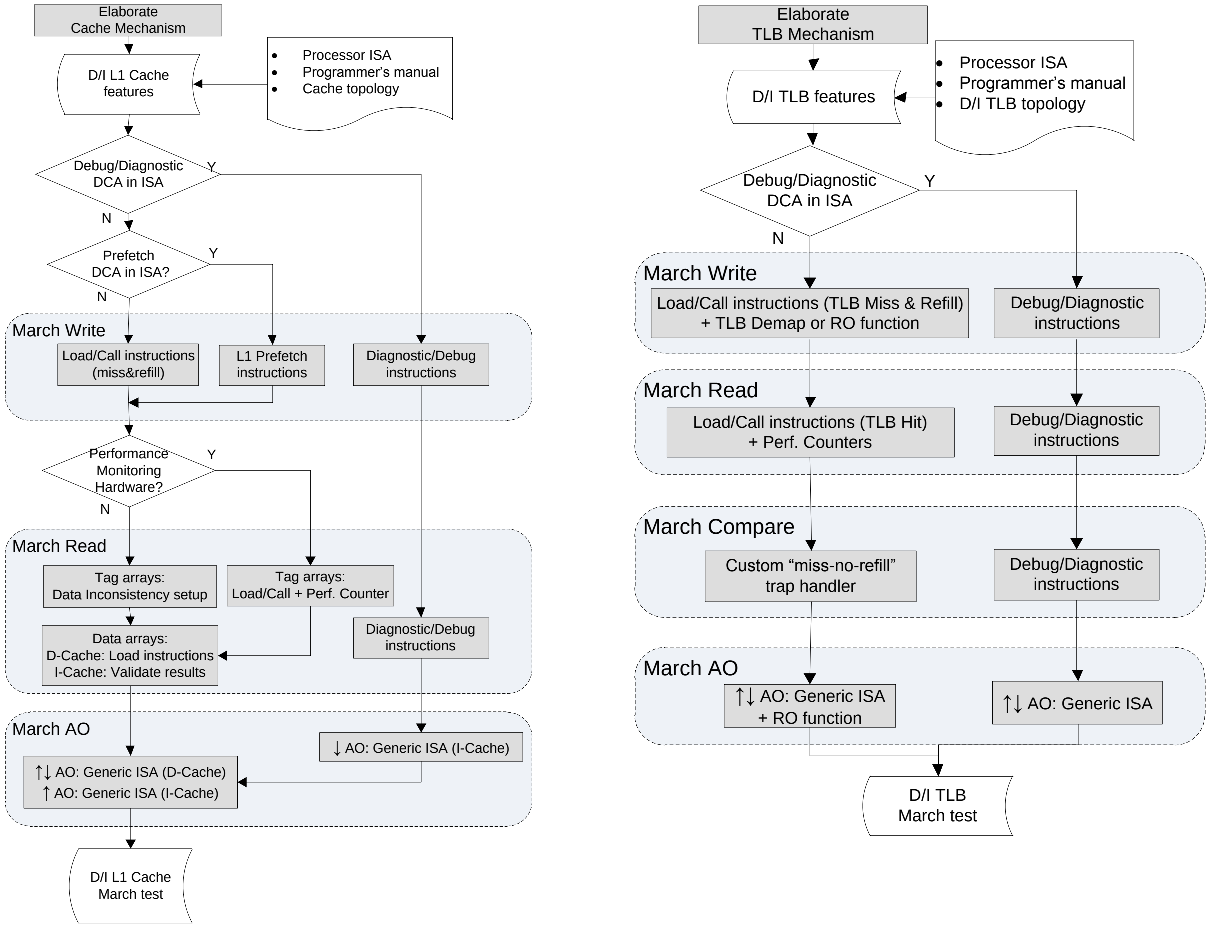

Elaborate Cache Mechanism
D/I L1 Cache features
Processor ISA
Programmer's manual
Cache topology
Debug/Diagnostic DCA in ISA
Y
N
Prefetch DCA in ISA?
Y
N
March Write
Load/Call instructions (miss&refill)
L1 Prefetch instructions
Diagnostic/Debug instructions
Performance Monitoring Hardware?
Y
N
March Read
Tag arrays: Data Inconsistency setup
Tag arrays: Load/Call + Perf. Counter
Data arrays: D-Cache: Load instructions I-Cache: Validate results
Diagnostic/Debug instructions
March AO
↓ AO: Generic ISA (I-Cache)
↑↓ AO: Generic ISA (D-Cache)
↑ AO: Generic ISA (I-Cache)
D/I L1 Cache March test
Elaborate TLB Mechanism
D/I TLB features
Processor ISA
Programmer's manual
D/I TLB topology
Debug/Diagnostic DCA in ISA
Y
N
March Write
Load/Call instructions (TLB Miss & Refill) + TLB Demap or RO function
Debug/Diagnostic instructions
March Read
Load/Call instructions (TLB Hit) + Perf. Counters
Debug/Diagnostic instructions
March Compare
Custom "miss-no-refill" trap handler
Debug/Diagnostic instructions
March AO
↑↓ AO: Generic ISA + RO function
↑↓ AO: Generic ISA
D/I TLB March test

Figure 26: SBST Methodology for a) L1 caches b) TLBs

The proposed SBST methodology implements low cost SBST March tests that target cache arrays by taking advantage of existing debug-diagnostic instructions in modern ISAs, as described in Section 4.2. These instructions must cover in total the fields of the ideal DCA instruction to overcome the testability challenges of the cache arrays. The proposed SBST methodology is summarized in Figure 26 for L1 caches and TLBs respectively. The main features of the proposed methodology are:

- Low cost March writes due to the high controllability of DCA write instructions.
- Low cost March reads due to the high observability of DCA read instructions.
- Low cost March comparisons due to the special features of DCA compare instructions.
- Low cost March reads for tag arrays by exploiting performance monitoring hardware, if DCA read instructions do not exist in the ISA.
- Low cost March comparisons for TLB tag arrays by exploiting the trap handler mechanism, if DCA compare instructions do not exist in the ISA.
- Test response compaction to comply with on-line testing requirements [41].

The methodology begins by extracting all available information about the cache architecture (e.g. L1 cache organization, TLB organization, common or distinct D/I TLB, size of the arrays, existence of control bits etc.), as well as, a thorough examination of the implemented ISA. This information is provided by the microarchitecture manual and the programmer's manual of the microprocessor.

A detailed description of the way that the methodology implements March operations for both data and instruction L1 cache and TLB arrays follows.

4.4.1 Data L1 cache

The methodology implements March element operations for the data L1 cache arrays as described in this section.

Write operations ($W_{DB}, W_{\overline{DB}}$ **)** are implemented by utilizing debug-diagnostic write instructions when available in the ISA. As above mentioned such instructions access both DL1-Tag and DL1-Data arrays and cover the desired fields (*WS*, *SS*, *WLS* and *A* fields) to initialize any cache line word from the lower memory hierarchy level (or the register file). This memory block (or register) must contain the test pattern to initialize either the DL1-Data array or the DL1-Tag array. If debug-diagnostic instructions do not exist in the ISA, L1 cache prefetch instruction is utilized for March write operation. Prefetch instructions initialize cache lines with blocks from memory. Tag is initialized with the high part of the addresses. Hence, a targeted selection of both the content and the address of the memory block can initialize both arrays with the desired test patterns. In case that DCA write instructions are not present in the ISA, generic load instructions for cache load miss and refill can be utilized to implement March write operation as proposed in [73] and [75].

Read operations ($R_{DB}, R_{\overline{DB}}$ **)** are implemented by utilizing debug-diagnostic read instructions when available in the ISA. These instructions access both DL1-Tag and DL1-Data arrays for read access in the same way that debug-diagnostic write instructions do (through *WS*, *SS*, *WLS* and *A* fields) to copy any cache line word to a memory location (or a register). Afterwards, the content of this memory location (or register) can be easily verified to complete March read operation. If debug-diagnostic instructions do not exist in the ISA, March read operations can be indirectly implemented by generic load instructions. In a load instruction, the expected test pattern in DL1-Data array (the one that was written by a March write operation) will be copied to a register where it can be verified. Furthermore, the same load instruction should produce a cache hit (or avoid a cache miss) if a successful March write operation in DL1-Tag has been preceded. This hit can be concurrently monitored by a performance counter with no extra cost, if available. In case that neither debug-diagnostic instructions nor performance counters are available, data inconsistency should be set up to validate March write operations for the D-Tag array as proposed in [73] and [75].

March addressing order (↑,↓) is implemented by utilizing generic ISA instructions. In data L1 cache, the implementation of both ascending and descending address order is straightforward. Generic instructions can be easily composed to implement a software loop to March DL1-Data and DL1-Tag arrays both in ascending/descending address order.

March SS for data L1 cache

```
// set associative : $N_d$ = # of sets , direct mapped : $N_d$ = # of lines
 $D = DB_{DATA}$
 $\overline{D} = \overline{DB}_{DATA}$
//M1 element ↑ ( $R_{DB}, R_{DB}, W_{DB}, R_{DB}, W_{\overline{DB}}$ )
for (i=0; i<$N_d$ ; i=i+1)                         // Access all sets/lines of cache
{
 A= create_address ( $DB_{TAG}$ : i : B )          //Address tag = $DB_{TAG}$
 $\overline{A}$ = create_address ( $\overline{DB}_{TAG}$ : i : B )          //Address tag = $\overline{DB}_{TAG}$
 DD _read ( A, D )                                 // or m x Load (A) + Perf Counter
 DD _read ( A, D )                                 // or m x Load (A) + Perf Counter
 DD _write ( A, D )                                // or Prefetch_block ( A )
 DD _read ( A, D )                                 // or m x Load (A) + Perf Counter
 DD _write ( $\overline{A}$ , $\overline{D}$ )                         // or Prefetch_block ( $\overline{A}$ )
 }
```

Figure 27: SBST routine for March SS test for data L1 caches

The proposed algorithmic notation of an SBST routine for both DL1-Data and DL1-Tag arrays of a data cache is shown in Figure 27. N_d is the number of cache sets (cache lines in case of a direct mapped organization) in a set-associative organization, *B* is the offset part of address to access a word inside the cache line, ($DB_{DATA}, \overline{DB}_{DATA}$) is the DB pair for the DL1-Data array, ($DB_{TAG}, \overline{DB}_{TAG}$) is the DB pair for the DL1-Tag array. Addresses *A* and $\overline{A}$ are created by the concatenation of the tag DBs, the parameter *I* (set/line index) and a value for *B* to access a word inside the cache line. Debug-diagnostic (DD) instructions as DCA instructions are used both for write and read operations when available. *DD_write(A,D)* instructions initialize a cache line/word in address A with content D. *DD_read(A,D)* instructions read a cache line/word in address A with content D. If such instructions do not exist in the ISA, a *Prefetch (A)* instruction is utilized to initialize an L1 cache line with the memory block that is located in address *A* and a sequence of *m* load instructions (*Load (A)*) is utilized to access all words in the cache line (*m* is the number of words per line). In every load instruction, a performance counter concurrently monitors for a miss to verify a successful March write on the DL1-Tag array, if available. If the debug-diagnostic instructions cannot access every cache way explicitly (lack of *WS* field), the routine is repeated *k* times for a *k*-way set-associative cache with LRU replacement with *k* different DB sets to cover the cache lines in every cache way.

The methodology can develop SBST routines for any March test. If debug-diagnostic instructions are utilized, the methodology can apply any DB including the most common ones that are used in the industry tests (solid, checkerboard, column stripes and row stripes) to both DL1-Data and DL1-Tag arrays. Otherwise, if such instructions do not exist, the DB pair ($\mathrm{DB}_{\mathrm{TAG}}, \overline{\mathrm{DB}}_{\mathrm{TAG}}$) should be defined in a memory data segment that is allowed by the virtual memory mechanism.

Finally, the proposed algorithmic notation can be easily simplified to test only the DL1-Tag array, if the DL1-Data array test is not required (e.g. a programmable MBIST scheme is present for the DL1-Data array).

4.4.2 Instruction L1 cache

Instruction L1 cache is used to store instructions fetched from lower levels of memory hierarchy. Even though instruction L1 cache is similar in structure to data L1 cache, the additional three testability challenges that have been described in Section 3 should be addressed.

The first challenge is due to the fact that the SBST routines affect the instruction L1 cache testing since they are also placed in the same cache during test application. Fetching the actual SBST routine in the instruction cache spoils the effectiveness of the test as the test patterns are substituted by the SBST code. In order to overcome this challenge, the SBST routine should be placed in a non-cacheable area of main memory. Alternatively, if the architecture cannot define a non-cacheable area, the cache enable/disable mechanisms can be used to isolate the SBST routine from cache. The utilization of the cache enable/disable mechanism increases the test code size and test time of the SBST routines. The rest of the challenges are overcome as described below.

The methodology implements March operations for the IL1-Data and IL1-Tag arrays, as follows:

Write operations ($W_{DB}, W_{\overline{DB}}$) are implemented by utilizing debug-diagnostic write instructions when available in the ISA. The instructions initialize both IL1-Data and IL1-Tag array similarly to data L1 cache arrays. Debug-diagnostic write instructions are not limited to utilize valid instructions for IL1-Data initialization, since the cache content must be invalidated when such instructions are executed. Hence, debug-diagnostic instructions overcome the testability challenge of composing test patterns for the IL1-Data array as they can initialize the IL1-Data array with any desired DB. If debug-diagnostic instructions do not exist in the ISA, L1 cache prefetch instruction is utilized for March write operation similarly to data L1 cache. When prefetch instructions are utilized, the test patterns should be valid instructions and are composed by instructions that have complementary opcode/fields. Every cache line should contain such valid instructions and the combination of them will form the desired test pattern. One return instruction should be included in every cache line to facilitate the return of the execution flow back to the SBST routine. When DCA write instructions are not present in the ISA, generic call instructions for L1 cache miss and refill can be utilized to implement March write operation as proposed in [73] and [79].

Read operations ($R_{DB}, R_{\overline{DB}}$**)** are implemented by utilizing debug-diagnostic read instructions when available in the ISA. These instructions copy the content of any L1 cache line word (IL1-Data) or any tag entry (IL1-Tag) to a memory location (or a register) by controlling the *WS, SS, WLS* and *A* fields, similarly to data L1 cache. Afterwards, the memory content (or register) can be easily verified to complete March read operation. The debug-diagnostic read instructions have high observability features since they implement direct access to the instruction L1 cache arrays that is missing through generic ISA instructions. If debug-diagnostic instructions do not exist in the ISA, a call instruction that targets the desired cache line should be utilized to fetch and execute all the instructions of the cache line. In order to verify successful March read operations for the IL1-Data, the executed instructions that have been used in the March write operation as test patterns should produce an unambiguous result that can be easily verified for its correctness as proposed in [79]. In order to verify successful March read operations for the IL1-Tag array, a call instruction should be executed to fetch instructions that are placed in the memory in a segment that is dictated by the desired test patterns (the high part of the addresses should compose the test pattern for the IL1-

Tag array). If a successful March write operation in IL1-Tag has been preceded, this call instruction will produce a cache hit (or avoid a cache miss) that can be directly monitored by a performance counter, if available. Note that a successful test will leave the performance counter's value to zero (zero cache misses) at the end of the test. In case that no performance monitoring hardware is available, data inconsistency should be set up to validate successful March write operations for the IL1-Tag array as described in [79].

March addressing order (↑,↓) is implemented by utilizing generic ISA instructions, when feasible, similarly to data L1 cache arrays. While the implementation of an ascending address order is straightforward, the implementation of a descending address order is challenging, because instructions are fetched only in ascending address order during normal system operation. When debug-diagnostic instructions are utilized, the high controllability and observability of these instructions allow a software routine to bypass the limitation of accessing cache lines in descending address order. A descending address order can be implemented by controlling the *WS, SS, WLS* and *A* fields to form a software loop. Moreover, when prefetch instructions are utilized and are combined with performance counters, a descending address order can be implemented for IL1-Tag by controlling the address *A* of the cache line to be accessed through a software loop.

The algorithmic notation of an SBST routine for both I-Data and I-Tag arrays of an instruction cache is shown in Figure 28. The symbol notations follow these of Figure 27.

```
March SS for instruction L1 cache
 // set-associative : Nd  = # of sets,  direct mapped : Nd   = # of lines
 $I = DB_{DATA}$
 $\overline{I} = \overline{DB}_{DATA}$
DC          //Disable cache (or placed in non-cacheable area)
//M1 element ↑ ( $R_{DB}, R_{DB}, W_{DB}, R_{DB}, W_{\overline{DB}}$ )
for (i=0; i<Nd ; i=i+1)                   // Access all sets/lines of cache
{
 A= create_address ( U1: i : B )      // Address to access test patterns
 $\overline{A}$ = create_address ( U2: i : B )      // Address to access test patterns
DD _read  ( A, I )                        // or EC; Call (A); DC+Perf. Counter
DD _read  ( A, I )                        // or EC; Call (A); DC + Perf. Counter
DD _write ( A, I )                        // or Prefetch_block ( A )
DD _read  ( A, I )                        // or EC; Call (A); DC + Perf. Counter
DD _write ( $\overline{A}$ , $\overline{I}$ )                        // or Prefetch_block ( $\overline{A}$ )
}
```

Figure 28: SBST routine for March SS test for instruction L1 caches

When debug-diagnostic instructions are present in the ISA, this template is valid for both IL1-Tag and IL1-Data. U1 and U2 upper parts of address are any valid memory segments that are initialized with the test patterns. On the contrary, when prefetch and call instructions are utilized, the selection of U1 and U2 memory blocks is more challenging. In order to apply March write and read operations to the IL1-Data array, these two memory blocks are initialized to contain cache lines with valid instructions that comply with the chosen ($DB_{DATA}, \overline{DB}_{DATA}$) pair. In this case, U1 and U2 can be any valid memory segment in the cacheable instruction segment. Each block should contain instructions that form the actual test pattern. One return instruction should be included in every cache line to facilitate the next March test iteration [79].

The verification of March read operations is performed by validating the execution result of the instruction sequence that forms the test patterns. If a ($DB_{DATA}, \overline{DB}_{DATA}$) pair cannot be defined in an ISA, two or more pairs of instructions with convenient formats (partially complementary adjacent bits) can be utilized ([72], [73]). In order to apply March writes and reads to the I-Tag array, U1 and U2 upper parts of address are the desired ($DB_{TAG}, \overline{DB}_{TAG}$) pair and must be selected carefully as in the majority of processor architectures the virtual memory mechanism does not allow to map instruction segments to any memory segment during on-line operation. The content of these blocks is not of high importance. The sole requirement is that every cache line should contain one return instruction in order to achieve continuous address order.

In the proposed template an enable/disable cache mechanism is utilized. Instruction cache is only activated, through the enable cache (EC) operation, when the call instruction is utilized in order to activate the performance counter's monitoring for the IL1-Tag.

4.4.3 TLBs

TLBs are small fully associative caches that store address related information in both their data and tag arrays. When a virtual address is generated by a store/load instruction it is stored to the DTLB-Tag array and its physical page number is issued to the DTLB-Data array. Respectively, when a virtual address is generated by an instruction fetch, it is stored to the ITLB-Tag array and its physical page number is issued to the ITLB-Data array. As described in Section 4.1.2, TLB arrays have two additional testability challenges: a) TLBs implement fully associative cache organization b) Tag arrays comprise a CAM memory that has to be also tested for CAM comparison faults. The methodology implements March operations (including March compare operation) for the TLB data and tag arrays, as follows:

Write operations ($W_{DB}, W_{\overline{DB}}$) are implemented by utilizing debug-diagnostic write instructions when available in the ISA as described above for L1 caches. Such instructions access both data and tag arrays of either D-TLB or I-TLB and cover the desired fields (*WS* and *A* fields) to initialize any TLB entry from the register file or the memory. This register (or memory block) must contain the desired DB to initialize either the data array or the tag array of the TLB under test. If debug-diagnostic write instructions do not exist in the ISA, March write operations can be implemented indirectly by a well-advised TLB miss through a generic store/load instruction for D-TLB or an instruction fetch for I-TLB on a preselected memory page for both data and tag array. The preselected memory page should have a physical address that contains the desired DB for the tag array and a virtual address that contains the desired DB for the tag array. In case of I-TLB arrays, these preselected memory pages should be initialized with one return instruction in order to facilitate the return of the execution flow back to the SBST routine. When TLB miss & refill mechanism is utilized, both data and tag arrays (DTLB-Data/DTLB-Tag or ITLB-Data/ITLB-Tag) can be written with the same instruction. However, serious limitations may arise due to LRU replacement policy. For example, in March SS algorithm in March element M_2, a W_{DB} operation cannot be directly applied after the R_{DB} through the miss and refill mechanism, as the desired TLB entry is already in the TLB array. According to this March element, it should be overwritten but it cannot be applied directly due to the LRU replacement policy. In order to overcome these difficulties the methodology exploits demap MMU entry operation when available in ISA or it utilizes an access reordering function (RO) - similar to the one that was presented in [79]- to force the LRU to implement an ascending/desceding address order.

Read operations ($R_{DB}, R_{\overline{DB}}$**)** are implemented by utilizing debug-diagnostic read instructions when available in the ISA. These instructions access both data and tag arrays of either the D-TLB or the I-TLB for read access through *WS,* and *A* fields to copy any TLB entry to a register (or a memory location). Afterwards, the content of this register (or memory location) can be easily verified to complete March read operation. If debug-diagnostic read instructions do not exist in the ISA, March read operations can be indirectly implemented by a well-advised TLB hit in a similar way that the March write operation was described above. TLB hits should be produced for every successful March read operation (March reads always verify March writes). These TLB hits or the total number of TLB misses - expected zero for a successful test - can be concurrently monitored by a performance counter with no extra cost. In case that neither debug-diagnostic instructions nor performance counters are available, March read validation can be indirectly performed by monitoring the increase in test execution time, due to unexpected D-TLB and I-TLB misses in case of defects in DTLB-Tag or ITLB-Tag arrays. Finally, defects in DTLB-Data and ITLB-Data arrays activate the page fault trap mechanism and validate the March test.

Compare operations *(*$C_{DB}, C_{\overline{DB}}$***)*** are implemented by utilizing debug-diagnostic compare instructions when available in the ISA. These instructions access the tag array and perform a CAM compare (also known as TLB entry match) operation. A register (or a memory location) is updated for either a D-TLB hit or a D-TLB miss (e.g. matched address or zero loads in Index register in MIPS R10000 when a *TLBP* instruction is executed). Hence, successful March compare operation can be easily validated afterwards. If debug-diagnostic compare instructions do not exist in the ISA, the methodology exploits the trap handler mechanism to implement a March compare operation. A custom “miss-no-refill” TLB trap handler can be defined to identify either a D-TLB or an I-TLB miss based on the native trap handler that serves TLB misses (the native trap handler performs a miss & refill operation). The custom trap handler should not refill TLB entries and can be used to indirectly implement March compares. Moreover, the handler routine should be programmed to count CAM misses to validate the March test.

March addressing order (↑,↓) is implemented by utilizing generic ISA instructions to control *WS* field when debug-diagnostic instructions are utilized as DCAs for March write, read and compare operations. If the ISA lacks such instructions and the native TLB miss & refill is utilized both ascending and descending address order are indirectly implemented by a RO function [79].

The proposed algorithmic notation of an SBST routine for I-TLB (both ITLB-Data and ITLB-Tag arrays) is shown in Figure 29. The symbol notations follow these of Figure 27 and Figure 28. March write and read operations are performed in a similar way with the one that is presented for L1 caches with either write/read debug-diagnostic instructions if present in ISA or with a well-advised *Call* instruction to produce a TLB miss/hit. DBs for TLB data and tag should be valid physical and virtual addresses when the ISA lacks debug-diagnostic instructions and the corresponding memory pages should contain a *return* instruction. *Pa2va* represents a function that performs physical to virtual address translation. In case of March compare operation, it is either implemented by a debug-diagnostic compare instruction *DD_compare (*$\overline{\mathrm{VA}}, \bar{\mathrm{I}}$*)* or by well advised *Call* instructions to produce a I-TLB miss that leads to the “miss-no-refill” trap handler mechanism.

```
March NCDA for I-TLB
//M5 element ↓ ( $R_{DB}, W_{\overline{DB}}, C_{\overline{DB}}$ )
for (i= $N_d$ ; i>0 ; i=i-1)      // $N_d$ = # I-TLB lines
{
  PA = ( $DB_{DATA}$ , i)                    // $DB_{DATA}$ are physical addesses
  $\overline{PA}$ = ( $\overline{DB}_{DATA}$ , i)
  VA = pa2va ( PA )                    // $DB_{TAG}$ are virtual addresses
  $\overline{VA}$ = pa2va ( $\overline{PA}$ )
  DD _read ( VA/PA, I )                //or Call (VA) + Perf. Counter
  DD _write ( $\overline{VA}/\overline{PA}$ , $\bar{I}$ )          //or Call ( $\overline{VA}$ )
  DD _compare ( $\overline{VA}$ , $\bar{I}$ )           //or Call( $\overline{VA}$ )+“miss-no-refill” trap handler
}
```

Figure 29: SBST routine for March NCDA for instruction TLBs

The methodology can develop SBST routines for any March test that targets the storage faults and the CAM oriented March tests (for tag array) that target comparison faults, excluding these March tests that utilize CAM masking [86], a CAM memory operation that cannot be implemented through ISA. If debug-diagnostic instructions exist in the ISA, the methodology can apply any DB and has no limitations for control bits that are included in the arrays. Otherwise, the DB composition is constrained by limitations in both the memory addressing schemes and the accessibility of control bits that lowers the effectiveness of the applied March tests. Finally, if D-TLB and I-TLB are unified - a common TLB architecture in modern processors - and the ISA lacks DCA instructions, it is preferable to apply the proposed SBST methodology and implement March test operations through load/store instructions (considering the unified TLB cache as a D-TLB cache during testing).

4.5 Multithreaded optimization for multi-bank L1 caches

In this section, we introduce the multithreaded optimization of our methodology. The proposed optimization elaborates the low level multiple bank organization of modern cache designs to exploit the thread level parallelism of modern multithreaded, multicore processors and speedup March test execution time.

In traditional cache designs, SRAM arrays are partitioned into multiple sub-banks to save power or to tweak the cache dimensions to fit smoothly in the given silicon real estate. These designs are called Uniform Cache Architectures (UCA). In modern cache designs, multibank Non-Uniform Cache Architectures (NUCA) are preferred to minimize internal wiring delay that are further divided in Static NUCA (S-NUCA) and Dynamic NUCA (D-NUCA) based on the way that the mapping of data into cache banks is achieved [87]. In this thesis, only UCA with multiple sub-bank organization and S-NUCA architectures are considered for the L1 cache SRAM arrays. Note that D-NUCA architectures are not used for L1 caches due to their small size.

In UCA and S-NUCA architectures, the mapping of data into cache banks and sub-banks is predetermined based on the block index of the architecture and thus can reside in only one bank of the cache. Several cache modeling tools (e.g. Cacti) enable fast exploration of the cache design space by automatically choosing the optimal bank and sub-bank count, size, and orientation of UCA and S-NUCA architectures. A typical S-NUCA cache physical organization that is adopted in modern processors -also considered by Cacti tool- is depicted in Figure 30.

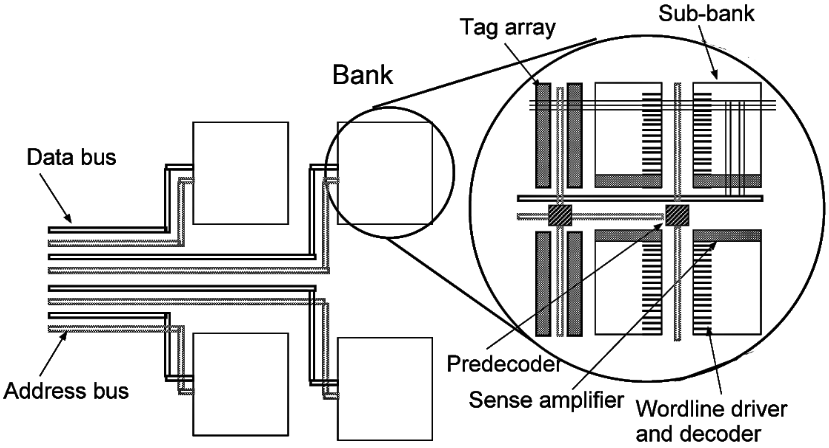


Figure 30: Cache multibank physical organization (S-NUCA)

The S-NUCA cache consists of multiple banks. Every bank is organized as a UCA with multiple sub-banks. A central pre-decoder drives signals to the local decoders of the sub-banks. Each sub-bank has a separate memory array, decoder, write drivers, and sense amplifier. As described above, the mapping of data into the banks and sub-banks is defined by the block index and is decided in the cache design process. Each cache bank or sub-bank in the above described UCA and S-NUCA multibank architectures can be considered as a separate SRAM array with a distinct functional fault set, since no coupling faults can occur between memory cells of such different cache banks and sub-banks. Therefore, an on-line testing strategy that considers every sub-bank as an independent memory array can be developed since the memory mapping is known to test engineers.

We propose the multithreaded optimization of our methodology that exploits the above mentioned feature of multibank cache organization. The proposed optimization considers exclusive fault sets for every L1 cache array sub-bank and proposes a fine-tuned clustering of the applied March tests in smaller subroutines based on the information provided for the physical implementation of a multibank cache (sub-bank address mapping, address scrambling etc.). Every cluster targets a sub-bank of the cache and the logical union of all clusters ends up to the initial March test for the whole cache array under test. These March test clusters target separate cache array sub-banks which can be executed concurrently without diminishing the March test effectiveness. Such a clustering is well suited to multithreaded processors, where concurrent execution can be achieved by assigning test clusters to different threads and hence SBST March test execution time is divided by the factor of available number of test threads. The test threads are dynamically scheduled software threads among the other executed processes. These test threads should be isolated during on-line testing in order to prevent the rest of the software processes to corrupt the March test effectiveness. In case that the March tests clusters outnumber the available threads, more than one March test clusters are assigned to every thread. Finally, the proposed multithreaded optimization is suitable both for simultaneous multithreading and interleaved multithreaded architectures, since it is independent of the way that the threads are issued and it is compatible with any resource allocation policy (e.g. physical register file size, register windows, register renaming e.t.c) that a multithreaded processor can implement.

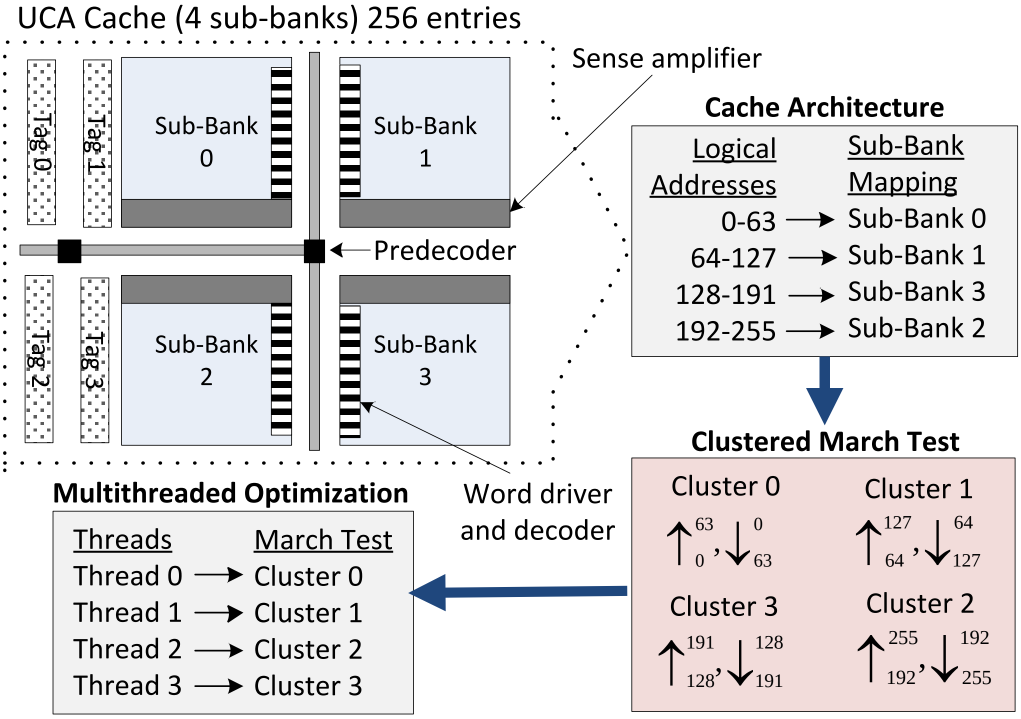


Figure 31: Example UCA cache – March Clustering

Example: Let us consider a processor that supports four threads per core. In Figure 31, we depict a UCA cache with four sub-banks (4x64=256 entries in total). The cache architecture determines the data mapping and every logical address can reside in only one sub-bank of the cache. In this example, gray code addressing is utilized for sub-bank allocation. Such an organization is very common to reduce switching activity in the cache predecoders. However the proposed optimization can be applied to any sub-bank addressing mode, since it is disclosed to test engineers. The sub-allocation addressing is elaborated to form the four March test clusters. Each cluster corresponds to the address subset of every sub-bank and both ascending/descending March address order are defined to form the SBST March test clusters by applying the proposed SBST March test development methodology. Afterwards, these four clusters are assigned to the processor four threads in order to be executed concurrently. Hence, each March test cluster is assigned to one thread, in order to achieve the maximum test time speedup.

5. CASE STUDIES

We have applied the proposed SBST methodology to three processor benchmarks a) LEON3, b) OpenRISC 1200 and c) OpenSPARC T1. We have also applied the multithreaded optimization to the L1 caches of OpenSparc T1.

In order to evaluate the effectiveness of the self-test routines we have used RAMSES memory fault simulator [88]. RAMSES consists of a simulation engine and numerous fault descriptors. Fault coverage is determined by evaluating the fault descriptors for predefined conditions. When coupling faults are concerned, the rest of the array cells except from the aggressor cell are possible victim cells. We have extended RAMSES to include fault descriptors on the basis of FPs [11] to include a) all the unlinked static storage faults [11] and b) CAM comparison faults [17] and we have implemented a test framework to bridge cache traces of Mentor Graphics' ModelSim and Synopsys VCS simulator with RAMSES to fault grade the cache arrays.

5.1 Benchmark 1 - LEON3

The first benchmark is LEON3, a publicly available processor designed by Aeroflex Gaisler that implements a SPARC V8 compliant architecture. We have configured the benchmark to include two 4KB 2-way set-associative L1 data and instruction caches with 64 sets and 8 words per cache line. This configuration includes two 64x29 tag arrays (D-Tag and I-Tag) and two 512x32 data arrays (D-Data and I-Data).

The SPARC V8 ISA, that LEON3 implements, includes privileged store/load instructions, denoted as alternate load/store (*lda/sta* instructions). These instructions can directly access cache arrays for diagnostic purposes by specifying alternate space identifiers (ASIs) that are defined by the SPARC architecture for both write and read access at supervisor level. These instructions have been used as DCA instructions for March write/read operations to apply and read the test patterns in SBST routines. In detail, alternate store (*sta*) instructions have been used to implement March write operations and alternate load (*lda*) instructions have been used to implement the March read operations. These instructions access all the cache SRAM arrays by utilizing the appropriate address indexing and the corresponding ASI. Note that when utilizing diagnostic accesses to a cache array, the cache should be invalidated afterwards. Hence, even in I-Data array, whose test patterns are formed by valid instructions, an SBST routine that takes advantage of alternate load instructions can apply any data background pair ($DB_{DATA}, \overline{DB}_{DATA}$) with no limitation because the cache will be invalidated after the test.

```
Element M1 of March SS: ↑(r0, r0, w0, r0,w1) for testing IL1-Data
// %r1 = DB vector   %r3 = 0x0 at the beginning of the test
//Disable ICACHE during test

2: lda [%r3] 0xd, %r5          //R0, I-Data content stored in %r5
   xor %r1, %r5,  %r5          //Compare with DB
   add %r7, %r5,  %r7          //Store validation result
   lda [%r3] 0xd, %r5
   xor %r1, %r5,  %r5
   add %r7, %r5,  %r7
   sta %r1, [%r3] 0xd          //W0, store %r1 to I-Data (%r3)
   lda [%r3] 0xd, %r5
   xor %r1, %r5,  %r5
   add %r7, %r5,  %r7
   sta %r2, [%r3] 0xd

   add  %r3,  4, %r3
   add  %r4,  4, %r4
   cmp  %r3, 2048
   bne  2b
```

Figure 32: Code snippet for LEON3's IL1-Data array

An assembly code snippet for March element M1 of March SS test for way 0 of I-Data array is shown in Figure 32. Instruction *lda [%r3] 0xd, %r5* fetches I-Data array contents of the address provided by *%r3* register (ASI 0xd is mapped to I-Data) to the general purpose register *%r5*. Instruction *sta %r1, [%r3] 0xd* initializes a cache content line in a similar way with a predefined value (the desired DB). Moreover, read validations are performed in every read operation and the validation of the result is compacted in register *%r7*. At the end of a successful test, the expected value of register *%r7* is zero. The same test with different address indexing targets way 1 of I-Data array. SBST routines for the rest of the cache arrays (D-Tag, D-Data, I-Tag) are formed in a similar way with different ASI values.

Table 8: LEON3 Data L1 Cache: SBST routines statistics

March test	Complexity (n)	DL1-Tag		DL1-Data	
		Test size (bytes)	Test Duration (cycles)	Test size (bytes)	Test Duration (cycles)
March C-	10n	384	6,360	384	47,300
March U	13n	384	6,680	384	50,340
March MSS	18n	512	10,160	512	77,100
March SS	22n	608	13,040	608	91,540

Table 9: LEON3 Instruction L1 Cache: SBST routines statistics

March test	Complexity (n)	IL1-Tag		IL1-Data	
		Test size (bytes)	Test Duration (cycles)	Test size (bytes)	Test Duration (cycles)
March C-	10n	428	42,060	428	310,140
March U	13n	424	43,220	424	340,060
March MSS	18n	536	68,460	536	539,180
March SS	22n	632	72,380	632	575,620

We have applied a set of March tests with different test complexities to both data and instruction caches. Solid data backgrounds (all-zero/all-ones) have been used to all tests. The test program statistics for both caches are shown in Table 8 and Table 9. The complexity of the March tests are expressed by their test lengths (n: number of bits of the array). The test routines are very effective in terms of test code size and test duration due to the utilization of the DCA instructions.

Finally, we evaluated test effectiveness of the test routines by utilizing RAMSES fault simulator. The port activity of cache arrays has been monitored and captured using ModelSim during the execution of SBST routines and then evaluated by RAMSES fault simulator for all unlinked static faults[1] to provide the achieved fault coverage. The coverage is complete (*100%*) for the fault models that every March test guarantees for both LEON3's data and instruction L1 cache SRAM arrays.

5.2 Benchmark 2 – OpenRISC 1200

The second benchmark is OpenRISC 1200, a publicly available processor core. The processor has been parameterized to utilize 4KB direct mapped write-through L1 caches that include 256x20 tag arrays (D-Tag & I-Tag) and 1024x32 data arrays (D-Data array & I-Data array). OpenRISC 1200 has been extended to include programmable performance counters that monitor data and instruction cache misses based on the specification of the designer's manual.

OpenRISC 1200 lacks debug-diagnostic instructions in its ISA to access the cache arrays. However, it includes a cache prefetch mechanism for both data and instruction L1 caches and maps prefetch operations to valid instructions. These instructions have been utilized as DCA instructions for March write operations. For March read operations, generic load and call instructions have been used. The observability of the March tests has been improved by exploiting the performance counters.

The cache prefetch operation in OpenRISC 1200 is implemented through a special purpose register (*DCBPR* register for data cache and *ICBPR* register for instruction cache). Indirect access to cache array has been used to implement March read operations by utilizing the load (*l.lwz*) instruction for the data cache and the jump and link register (*l.jalr*) instruction for the instruction cache. An enable/disable cache mechanism has been utilized since OpenRISC 1200 lacks the ability to define a non-cacheable area. In D-Data array, read validation has been performed by comparing the *l.lwz* instruction result with a golden value in every March test iteration. In I-Data array, two instructions with complementary opcodes (a register vector addition *lv.adds.h* and the immediate store *l.sw*) have been used to form the test patterns. At the end of every cache line we have placed a jump register (*l.jr*) instruction. The March read operations have been validated by elaborating the result of both *lv.adds.h* and *l.sw* execution. We

[1] **Single Cell Faults**: State (SF), Transition (TF), Write Destructive (WDF), Read Destructive (RDF), Deceptive Read Destructive (DRDF), Incorrect Read (IRF)

Cell Coupling Faults (2-cell): State (CFst), Disturb (CFds), Transition (CFtr), Write Destructive (CFwd), Read Destructive (CFrd), Deceptive Read Destructive (CFdr), Incorrect Read (CFir) have executed twice the same routines for I-Data array. The first execution with the *l.jr* instruction located at the last word of every cache line and the second one with the *l.jr* instruction located to another word inside the cache line to detect the remaining faults that are masked by the utilization of a jump instruction [79]. Cache misses monitored by performance counters for D-Tag and I-Tag arrays verify March read operations at the end of the test. An assembly code snippet for March SS M1 element for D-Tag array is shown in Figure 33.

Element M1 of March SS*: ↑(r0, r0, w0, r0,w1) for DL1-Tag*

```
// (r1)   = 0x00000000 at the beginning of the test
// (r30) = 0xFFFFF000 at the beginning of the test
l.mtspr r0,r11,SPR_PCMR(0)              //Enable performance counter

march_loop_m1:
l.lwz r5,0(r1)                         //R0, if cache miss, counter ++
l.lwz r5,0(r1)                         //R0, if cache miss, counter ++
l.mtspr r0,r1,SPR_DCBPR                //W0, prefetch block
l.lwz r5,0(r1)                         //R0, if cache miss, counter++
l.mtspr r0,r30,SPR_DCBPR               //W1, prefetch block
l.addi  r1,r1,16                       //loop control
l.sfeqi  r30,4080
l.bnf   march_loop_m1
l.addi  r30,r30,16
l.mtspr r0,r12,SPR_PCMR(0)              //Disable performance counter
```

Figure 33: Code snippet for OpenRISC's 1200 DL1-Tag array

We have implemented the same March tests that we presented for LEON3. The test program statistics for both caches are shown in Table 10 and Table 11. As shown in tables, test programs, are cost-effective both in terms of code size and test duration. Routines for instruction cache have longer test duration as they have been executed with the instruction cache disabled. Moreover, test duration is even longer for the I-Data array because the test routines were executed twice.

Table 10: OpenRISC 1200 Data L1 Cache: SBST routines statistics

March test	Complexity (n)	DL1-Tag		DL1-Data	
		Test size (bytes)	Test Duration (cycles)	Test size (bytes)	Test Duration (cycles)
March C-	10n	280	16,667	1,016	58,217
March U	13n	288	18,000	1,436	81,060
March MSS	18n	344	22,666	1,496	95,023
March SS	22n	384	27,000	2,200	147,129

Table 11: OpenRISC 1200 Instruction L1 Cache: SBST routines statistics

March test	Complexity (n)	IL1-Tag		IL1-Data	
		Test size (bytes)	Test Duration (cycles)	Test size (bytes)	Test Duration (cycles)
March C-	10n	744	658,334	800	1,585,836
March U	13n	780	770,666	832	1,811,222
March MSS	18n	893	970,420	1,120	2,540,104
March SS	22n	1,128	1,180,400	1,296	3,185,278

Finally, we have evaluated test effectiveness of the test routines by utilizing RAMSES fault simulator in a similar way with the one that is presented for LEON3 processor. The achieved fault coverage is complete (100%) for all the above mentioned fault models that every March test guarantees for the OpenRISC's data cache. For the OpenRISC's instruction cache, the fault coverage is complete (100%) for all single cell faults and 99% for the coupling faults. The fault coverage for the coupling faults is slightly lowered due to the utilization of the enable/disable mechanism instead of a non-cacheable area which is not supported. The *l.jalr* instruction (March read operation) is executed with the instruction cache enabled to activate the performance counter. The corresponding memory block that is fetched to the instruction cache corrupts a single test pattern and slightly lowers the fault coverage. If complete fault coverage is required for coupling faults, the March test can be fetched and partially executed again from another memory segment to complete the missing fault coverage as a tradeoff to increased test code size and test duration.

5.3 Benchmark 3 – OpenSPARC T1

The third benchmark is OpenSPARC T1 that includes both data and instruction L1 caches and fully functional TLBS. Therefore, the SBST methodology has been fully applied to both the L1 cache and TLB arrays.

OpenSPARC T1 is a multithreaded chip multiprocessor with eight SPARC V9 processor cores. Each SPARC core has hardware support for four hardware threads. T1 utilizes thread level parallelism (TLP) in a fine grain multithreading mechanism. Each SPARC core has separate data and instruction L1 caches: a 4-way set-associative 8 KB L1 data cache and a 4-way set-associative 16 KB L1 instruction cache. The cache line size is 16 bytes for the data cache and 32 bytes for the instruction cache. The cache SRAM arrays are organized as follows: the DL1-Data array is physically implemented as two 128x288 SRAM banks, the IL1-Data array is implemented as four 128x272 SRAM banks. Both the DL1-Tag and IL1-Tag array are physically implemented as two 64x132 banks. The control bits for both data and instruction L1 caches are not part of the SRAM arrays. They are implemented as a separate register array.

Additionally, each SPARC core has a separate D-TLB and I-TLB. Both D-TLB and I-TLB consist of a 64-entry SRAM memory (data array) and a 64-entry fully associative CAM memory (tag array). The data entries are 43 bits wide and the tag entries are 59 bits wide. The Physical Page Number (PPN) width is 27 bits. The rest of the bits in data array entries are control bits (16 bits). The Virtual Page Number (VPN) width is 35 bits. Moreover, a 13-bit context field and a 3-bit partition field are included. The rest of the bits in the tag array are control bits.

OpenSPARC T1 implements a SPARC V9 compliant ISA and includes privileged store/load instructions, denoted as alternate load/store (*ldxa/stxa* instructions). These instructions can directly access all cache arrays for debug/diagnostic purposes by specifying alternate space identifiers (ASIs) that are defined by the SPARC architecture for both write and read access at supervisor level. We have exploited these alternate load/store instructions for March write/read operations to directly access all cache arrays for both L1 caches and TLBs for March write and March read operations at low cost by utilizing the appropriate ASI at the hypervisor level. Moreover, OpenSPARC T1 implements performance monitoring hardware (performance counters) that can be configured to monitor for L1 cache and TLB misses. In order to further optimize the test execution time we combined the utilization of performance counters for the both L1 cache tag arrays (DL1-Tag and IL1-Tag), even though DCA read instructions could be utilized. By combining the DCA write instructions with performance counters for the cache tag arrays we have achieved to validate the whole test for tag arrays with a single step by verifying the performance counter's value at the end of the test (expecting to be zero). SPARC V9 ISA does not implement a debug/diagnostic compare instruction for implementing the March compare operation. Therefore, a custom “miss-no-refill” trap handler has been utilized to implement March compare operations to test DTLB-Tag and ITLB-Tag CAM arrays for comparison faults.

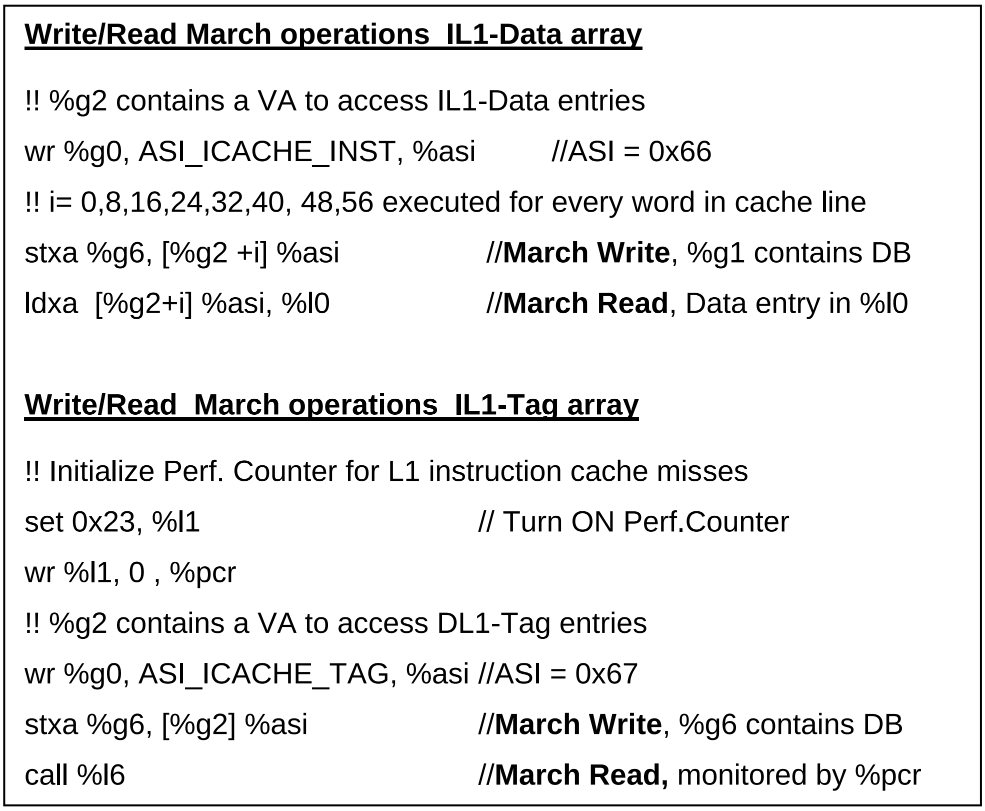
Write/Read March operations DL1-Data array

```
!! %g2 contains a VA to access DL1-Data entries
wr   %g0, ASI_DCACHE_DATA, %asi             //ASI = 0x46
stxa %g1, [%g2] %asi                 //March Write, %g1 contains DB
stxa %g1, [%g2+8] %asi               // Second word in cache line
ldx  [%g5], %l0                      //March Read, Data entry in %l0
ldx  [%g5+8], %l1                    // Second word stored in %l1
```

Write/Read March operations DL1-Tag array

```
!! Initialize Perf. Counter for L1 data cache misses
set  0x33, %l0
wr  %l0, 0 , %pcr
!! %g2 contains a VA to access DL1-Tag entries
wr   %g0, ASI_DCACHE_TAG, %asi               //ASI = 0x47
stxa %g6, [%g2] %asi                 //March Write, %g1 contains DB
ldx [%g5], %l0                       //March Read, monitored by %pcr
```

Figure 34: Code snippet for OpenSPARC T1 L1 Data cache



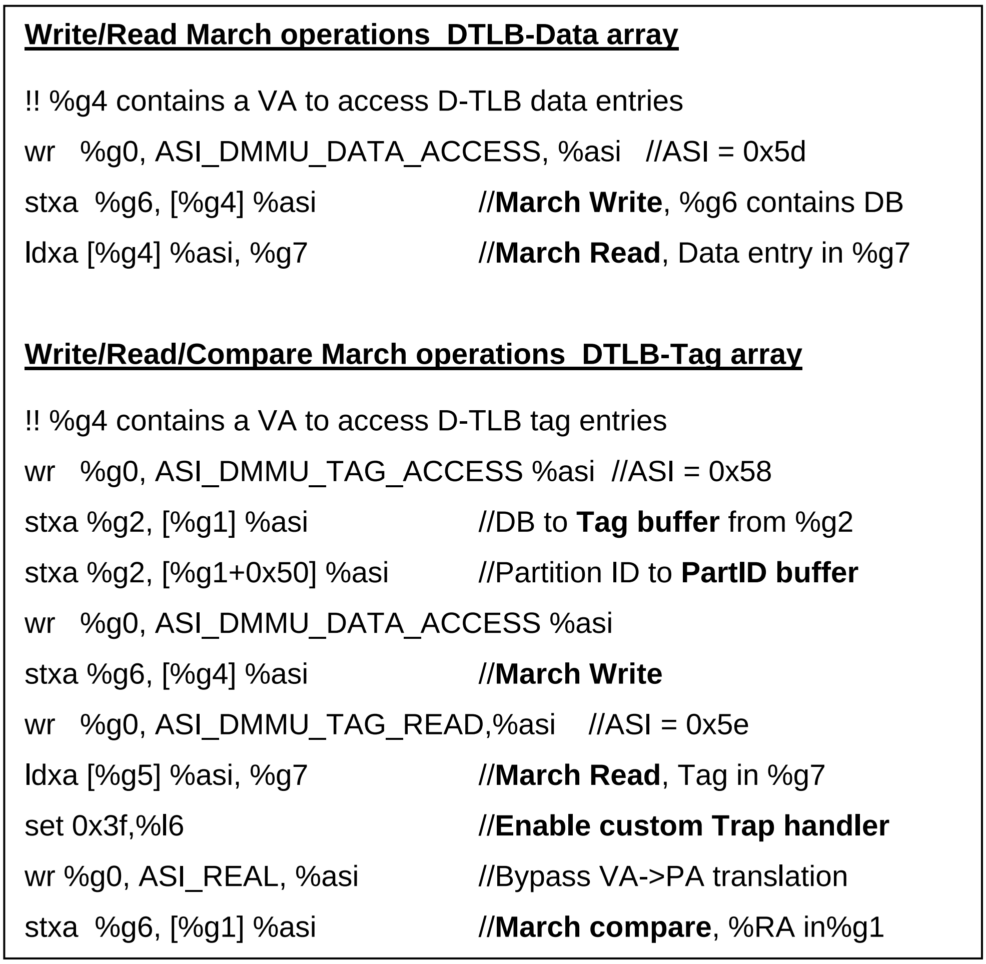
Write/Read March operations IL1-Data array

```
!! %g2 contains a VA to access IL1-Data entries
wr %g0, ASI_ICACHE_INST, %asi         //ASI = 0x66
!! i= 0,8,16,24,32,40, 48,56 executed for every word in cache line
stxa %g6, [%g2 +i] %asi               //March Write, %g1 contains DB
ldxa  [%g2+i] %asi, %l0               //March Read, Data entry in %l0
```

Write/Read March operations IL1-Tag array

```
!! Initialize Perf. Counter for L1 instruction cache misses
set 0x23, %l1                         // Turn ON Perf.Counter
wr %l1, 0 , %pcr
!! %g2 contains a VA to access DL1-Tag entries
wr %g0, ASI_ICACHE_TAG, %asi //ASI = 0x67
stxa %g6, [%g2] %asi                  //March Write, %g6 contains DB
call %l6                              //March Read, monitored by %pcr
```

Figure 35: Code snippet for OpenSPARC T1 L1 Instruction cache



Write/Read March operations DTLB-Data array

```
!! %g4 contains a VA to access D-TLB data entries
wr   %g0, ASI_DMMU_DATA_ACCESS, %asi   //ASI = 0x5d
stxa  %g6, [%g4] %asi                 //March Write, %g6 contains DB
ldxa [%g4] %asi, %g7                  //March Read, Data entry in %g7
```

Write/Read/Compare March operations DTLB-Tag array

```
!! %g4 contains a VA to access D-TLB tag entries
wr   %g0, ASI_DMMU_TAG_ACCESS %asi  //ASI = 0x58
stxa %g2, [%g1] %asi                  //DB to Tag buffer from %g2
stxa %g2, [%g1+0x50] %asi             //Partition ID to PartID buffer
wr   %g0, ASI_DMMU_DATA_ACCESS %asi
stxa %g6, [%g4] %asi                  //March Write
wr   %g0, ASI_DMMU_TAG_READ,%asi    //ASI = 0x5e
ldxa [%g5] %asi, %g7                  //March Read, Tag in %g7
set 0x3f,%l6                          //Enable custom Trap handler
wr %g0, ASI_REAL, %asi                //Bypass VA->PA translation
stxa  %g6, [%g1] %asi                 //March compare, %RA in%g1
```

Figure 36: Code snippet for OpenSPARC T1 D-TLB

Write/Read March operations ITLB-Data array

```
!! %g4 contains a VA to access I-TLB data entries
wr %g0, ASI_IMMU_DATA_ACCESS, %asi   //ASI = 0x55
stxa %g6, [%g4] %asi                 //March Write, %g6 contains DB
ldxa [%g4] %asi, %g7                 //March Read, Data entry in %g7
```

Write/Read/Compare March operations ITLB-Tag array

```
!! %g4 contains a VA to access I-TLB tag entries
wr %g0, ASI_IMMU_TAG_ACCESS %asi       //ASI = 0x50
stxa %g2, [%g1] %asi                 //DB to Tag buffer from %g2
stxa %g2, [%g1+0x50] %asi            //Partition ID to PartID buffer
wr %g0, ASI_IMMU_DATA_ACCESS %asi
stxa %g6, [%g4] %asi                       //March Write
wr %g0, ASI_IMMU_TAG_READ,%asi             //ASI = 0x56
ldxa [%g5] %asi, %g7                       // March Read, Tag in %g7

wr %g0, ASI_LSU_CONTROL_REG, %asi
set  0xB, %l6                        //Bypass VA->PA translation
stxa %l6, [%g0] %asi                 //for instruction fetch
set 0x3f,%l6                         //Enable custom Trap handler
jmpl %o2, %o4                        //March compare,%RA in %o2
```

Figure 37: Code snippet for OpenSPARC T1 I-TLB

Assembly code snippets that show how March operations are implemented for all cache arrays (both L1 caches and TLBs) are shown in Figures 34-37. In detail, we have utilized alternate store (*stxa*) instruction to implement March writes in all cache arrays and utilized alternate load (*ldxa*) instructions to implement March reads for IL1-Data array and all TLB arrays in order to overcome the testability challenges for implementing March read operations. On both L1 cache tag arrays (DL1-Tag and IL1-Tag) we have exploited the performance counters to monitor for cache misses for validation of March read operations that are performed by load instructions in DL1-Tag array and call instructions in IL1-tag array, respectively. Note that March write operation for TLB tag arrays is implemented by utilizing three *stxa* instructions. Tag entry (VPN, context ID and control bits) and Partition ID must firstly be placed in the Tag buffer and PartID buffer, and then March write operation for tag array is applied. The trap handler that we have programmed to implement March compare operation differs to the native one that handles TLB misses in two aspects: a) It does not refill the TLB when a miss occurs, b)

It can access physical addresses beyond the 8GB limit (which is an OpenSPARC T1 convention for memory address space). This handler is enabled by utilizing a predefined value in register *%l6* and when either a generic store/load instruction occurs for DTLB-Tag or instruction fetch for ITLB-Tag it detects hits and misses without refilling any TLB entry. Real addresses were utilized for both DTLB-Tag and ITLB-Tag to avoid generating page faults, for non-permitted physical address during testing. March compare is implemented by the *ldxa* instruction for the DTLB-Tag and by the *jmpl* instruction for the ITLB-Tag in order to cause a “miss-no-refill” to the D-TLB and I-TLB respectively.

We have implemented the same set of contemporary March tests that we have implemented for the LEON3 benchmark for storage faults [11]. March C, March MSS and March SS are applied to all cache arrays (both for L1 caches and TLBs) to detect storage faults. Moreover, we have implemented a set of contemporary March tests for comparison faults [17]. The March test of [89] is applied to DTLB-Tag and ITLB-Tag array only and detects both storage and comparison faults (March C- is included in this test), while March CFT [90] is also applied to these arrays but targets comparison faults only. The implemented SBST routines were stored and fetched from OpenSPARC T1’s main memory.

We have also applied the multithread optimization to the March tests that target the four SRAM arrays of both data and instruction L1 caches of a SPARC core. At first, every March test has been executed by a single thread of the SPARC core and the effectiveness of the test has been evaluated in terms of test code size and test execution time. Afterwards, we elaborated the cache architecture and design (based on the designer’s manual) for optimizing the test execution time by splitting every implemented March test to smaller clusters (based on sub-bank cache organization) and assign each cluster to a processor thread based on the optimization methodology that was described in Section 4.5. The DL1-Data, DL1-Tag and IL1-Tag cache arrays are organized in two sub-banks and the test routines were split into two threads. The IL1-Data array consists of four sub-banks and therefore the test routine was split to four threads, respectively.

The statistics for both the L1 caches and TLBs are shown in Tables 12-15. The test codes that target the L1 Data cache arrays and the D-TLB arrays have been unified to further optimize the test execution time whereas the test codes that target the L1 instruction cache arrays and the I-TLB arrays have been executed separately. The complexity of the March tests are expressed by their test lengths (n: total number of bits of the array, L: number of bits of TLB tag line). Note that the test routines are very effective in terms of test code size and test time due to the utilization of the DCA instructions.

When the multithreaded optimization is applied, the test execution time speedup is about 1.7 when executed in two threads (D-Tag, D-Data & I-Tag) and about 3.7 when executed in four threads (I-Data only). The thread level parallelism of March tests has a negative impact on test code size because the code has to be partially duplicated or even quadruplicated to be executed by different threads.

Table 12: OpenSPARC T1 data L1 cache: SBST routines statistics

		DL1-Tag & DL1-Data				
		1 thread		2 threads		
	Complexity (n)	Test size (bytes)	Test Time (cycles)	Test size (bytes)	Test Time (cycles)	Speedup
March C-	10n	960	213,164	1,512	124,588	1.71
March U	13n	972	271,844	1,656	155,548	1.75
March MSS	18n	1,216	354,424	2,012	201,432	1.76
March SS	22n	1,344	387,932	2,264	222,976	1.74

Table 13: OpenSPARC T1 instruction L1 cache (IL1-Tag): SBST routines statistics

		IL1-Tag				
		1 thread		2 threads		
	Complexity (n)	Test size (bytes)	Test Time (cycles)	Test size (bytes)	Test Time (cycles)	Speedup
March C-	10n	1,016	1,883,524	1,772	1,173,524	1.60
March U	13n	1,028	2,224,052	1,864	1,378,904	1.61
March MSS	18n	1,256	2,869,900	2,192	1,638,896	1.75
March SS	22n	1,536	3,335,784	2,472	1,924,392	1.73

Table 14: OpenSPARC T1 instruction L1 cache (IL1-Data): SBST routines statistics

		IL1-Data				
		1 thread		4 threads		Speedup
	Complexity (n)	Test size (bytes)	Test Time (cycles)	Test size (bytes)	Test Time (cycles)	
March C-	10n	1,392	3,065,184	4,140	817,696	3.75
March U	13n	1,508	3,633,048	4,764	973,928	3.73
March MSS	18n	2,048	5,067,132	6,700	1,358,012	3.73
March SS	22n	2,560	6,468,283	8,748	1,746,432	3.70

Table 15: OpenSPARC T1 TLBs: SBST routines statistics

March test	Complexity (n)	DTLB		ITLB	
		Test size (bytes)	Test Duration (cycles)	Test size (bytes)	Test Duration (cycles)
March C-	10n	424	26,412	768	49,784
March MSS	18n	520	33,972	956	66,948
March SS	22n	684	49,240	1,050	85,476
March of [89]	14n+2L	712	42,984	1,160	60,164
March CFT [90]	5n+2L	556	22,764	636	36,608

Finally, we evaluated the test effectiveness of the SBST routines with the in-house developed extended version of RAMSES fault simulator. The achieved fault coverage is complete (100%) for all the storage faults that every March test guarantees for all four L1 cache arrays (DL1-Data, DL1-Tag, IL1-Data and IL1-Tag) and all four TLB arrays (DTLB-Data, DTLB-Tag, ITLB-Data and ITLB-Tag) when debug-diagnostic instructions are exploited. For CAM comparison faults[2], the fault coverage is slightly lowered to 91-92% for both DTLB-Tag and ITLB-Tag arrays due to the lack of a debug-diagnostic

[2] **Comparison Faults**: Stuck-Match (SMF), Stuck-MisMatch (SMMF), Conditional-Match (CMF), Partially-Match (PMF), Equivalence-MisMatch (EMMF), Inequivalence-Match (IMF), Cross-Match (XMF), Cross-Mismatch (XMMF)

compare instruction and the utilization of the custom trap handler that addresses limitations in accessing the control bits.

5.4 Performance overhead evaluation

In this section, we present the evaluation framework that was utilized to estimate the performance overhead of the proposed SBST routines. We have implemented several on-line periodic testing scenarios and we will present detailed statistics of the performance overhead introduced in a typical workload under these test scenarios.

We have utilized a SunFire T2000 server running a set of multithreaded programs -the PARSEC benchmark suite- over Solaris 10 to evaluate the performance overhead of the deployed SBST routines. Our server is powered by a quad-core UltraSPARC T1 processor. OpenSPARC T1 processor is the free version of UltraSPARC T1 that is utilized in SunFire T2000 servers. Hence, the SBST routines that were developed above for OpenSPARC T1 processor can be directly compiled to our server to evaluate their performance overhead.

We have selected the optimized 2-thread March C- SBST routine that targets the data L1 cache (both DL1-Tag and DL1-Data) to be utilized as our self-test routine in the evaluation framework. The statistics of this test routine are shown in the 1st line of Table 12 and its test execution time has been measured about 1.2sec in our system. Any other SBST routine (or a set of March tests) could have been selected. Since we do not have access on the hypervisor level on the UltraSPARC T1 processor of a native system, we have slightly altered the SBST test routines in order to comply with Solaris OS limitations on executing hyper privileged instructions.

The modified SBST routines have the same memory footprint and test execution time with the actual one that was presented in the previous subsection, thus the modified self-test routine is sufficient for studying the performance impact of the proposed on-line self-tests.

Here after, we will utilize the terms of Test Period (TP) and Test Latency (TL) as described below:

- Test period (TP) and is the amount of time from the beginning of a self-test on a core to the beginning of the next self-test on the same core.
- Test latency (TL) is the duration of an on-line self-test.

We have applied several on-line testing scenarios with a fixed TL (1.2sec) and several short TPs (< 1min) that are suitable for detecting early-life failures on two different framework configurations, a *1core/4thread* setup and a *4core/16thread* setup as described below in detail.

- ***1core/4threads setup, TL=1.2sec, TP=2, 15 and 60 sec***

Firstly, Solaris capability of creating virtual processor sets has been exploited to isolate a single SPARC core from OpenSPARC T1 and both PARSEC workload applications and SBST routine have been set -by utilizing Light Weight Process (LWP) binding- to be executed in this SPARC core.

Note that we have selected to isolate an idle core that does not execute any other OS process in order to evaluate the real performance overhead due to the SBST routine's periodic execution only. The PARSEC suite has been configured to be executed by four threads (the maximum number of threads in the core). These four threads share the same L1 cache. Hence, both workload and March test application access the same cache SRAM arrays. Afterwards, several TPs have been selected to represent different test scenarios. For example a demanding testing scenario may require intensive test period (e.g. TP=2sec) while a more relaxed test scenario may require less intensive test periods (e.g. TP=60sec).

- ***4cores/16threads setup, TL=1.2sec, TP=10, 30 and 60 sec***

In this configuration we have utilized all four available SPARC cores of our server. Hence, apart from the PARSEC applications and the periodic execution of the SBST routine, the OS processes were also executed in the background. The PARSEC suite has been configured to be executed by all the available sixteen threads of our system (four threads per SPARC core). A script has been composed to call the SBST routine for every SPARC core in a round-robin way in every TP. Moreover, in each SBST routine execution, the script forms a virtual processor set of the 4 threads that belong to the core under test to ensure that the SBST routine will be executed only by the selected core under test. This is a critical requirement to guarantee the test quality by preventing the test patterns to be stored in L1 cache of other cores, apart from the core under test. Several TPs have been selected to represent different test scenarios, in a similar way that was presented for the *1core/4threads* setup. The selected TPs were longer in these case studies due to the need of executing the same SBST routine four times (one for the L1 cache of each SPARC core).

The system in both configuration setups was configured to execute all the 13 multithreaded programs of the PARSEC suite. All PARSEC programs have been compiled to utilize the pthreads parallelization model and the native dataset was utilized in all simulations. After estimating the workload execution time in both configuration setups without test, the PARSEC programs were executed several times while the SBST routine was scheduled to be executed in the background at a fixed TP in every test scenario for both configurations.

In Figure 38 and Figure 39 we present the performance overhead of the PARSEC applications caused by the periodic execution of the SBST March test in all the periodic test scenarios that have been examined for both framework setups. Note that the performance overhead for each PARSEC application differs significantly. Some of the applications have long execution time (e.g. blackscholes and x264) with large input datasets and their performance was severely degraded by the SBST periodic execution, while other applications have shorter execution time (e.g. freqmine and dedup) with smaller data sets and their performance was slightly degraded.

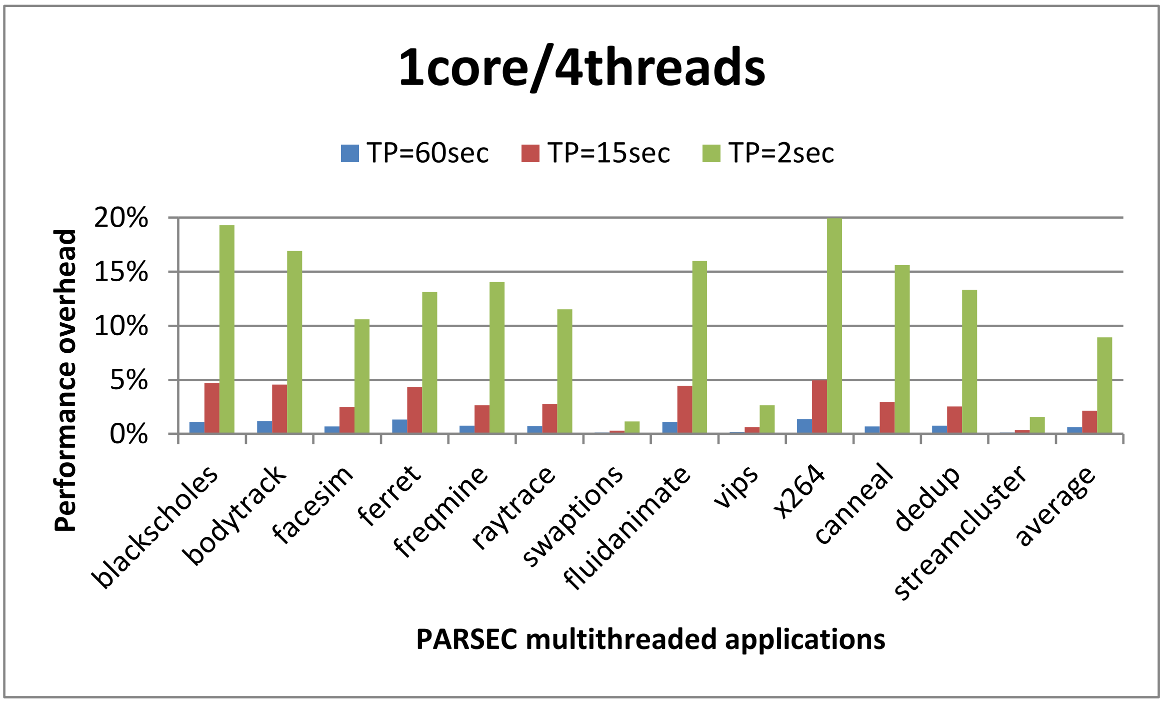


Figure 38: Performance overhead for PARSEC workload (1core/4threads)

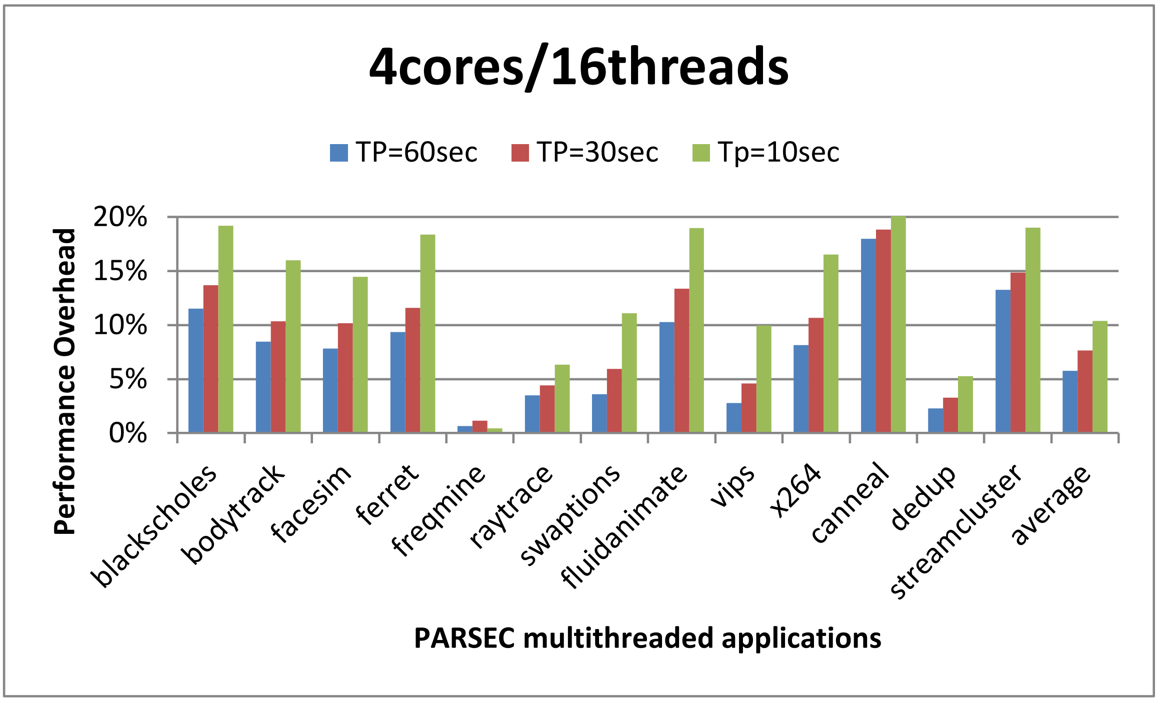


Figure 39: Performance overhead for PARSEC workload (4cores/16threads)

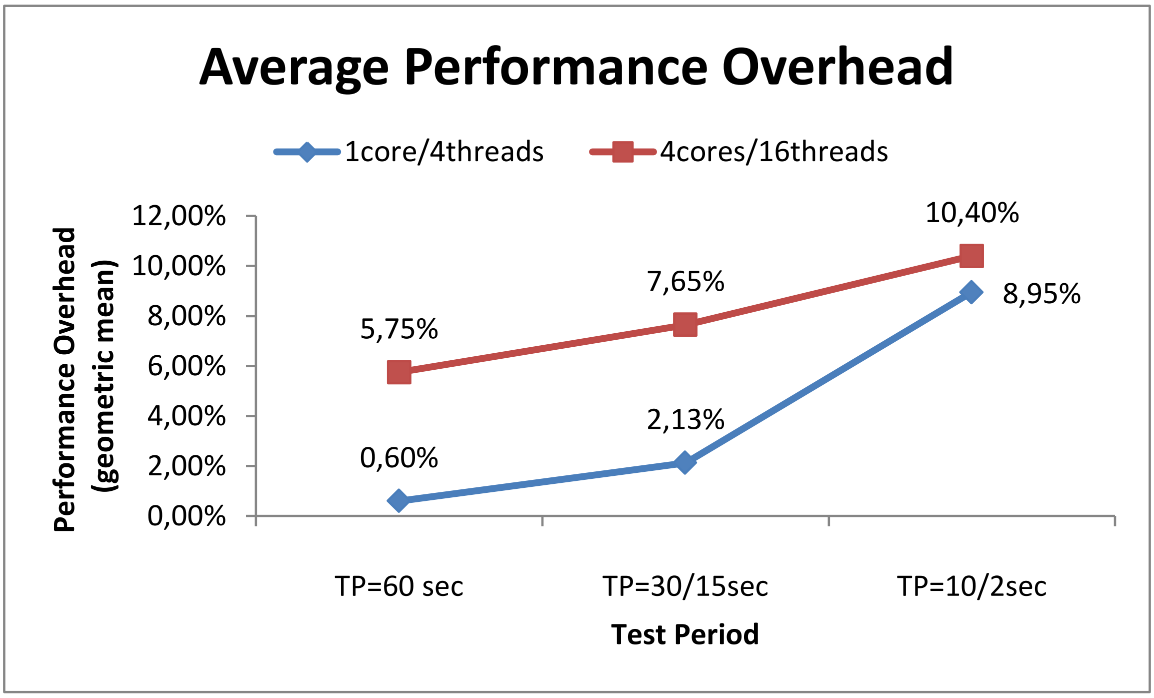


Figure 40: SBST routines performance overhead

The geometric mean of all the PARSEC applications in every test scenario is shown in the last column (average) of both setup configurations and are also presented in Figure 40 for both configuration setups.

In both configuration setups, the performance overhead increases in smaller test periods as the self-test routine is executed more frequently during the PARSEC suite execution. Moreover, the performance overhead for the quad core setup is higher than the one that occurs in the single core setup in all test scenarios because of the need to test every L1 cache (every core has a dedicated L1 cache) for every core in every test period. However, based on the experimental results, the performance overhead even in the more demanding test scenario of a quad core processor does not exceed the 11% of the performance of the system without test. Considering that this performance penalty refers to a demanding on-line periodic self-test scenario that applies March C- algorithm to both D-Tag and D-Data SRAM arrays for all four L1 caches (each data cache has a size of 8KB in a SPARC core) of the quad core system every 10sec, such performance degradation can be affordable when a high frequency test scenario is required. In contrary, in a relaxed test scenario (e.g. not more than a test per minute), the performance overhead is lower. For example, when the SBST routine is periodically executed every minute in a single core system (e.g. a single core system can be an embedded processor), the performance overhead is less than 1%, thus negligible. Thus, the proposed SBST methodology can periodically apply March tests to L1 caches effectively during the system’s lifetime with acceptable performance overhead in workload execution.

6. COMPARISONS

In this section, we compare our methodology that exploits DCA instructions to implement SBST March tests against previous approaches that provide pseudo-instruction sequences ([72] and [75]) and experimental results of the March operations ([73], [75] and [79]) for cache arrays only. In order to directly compare the proposed methodology that exploits debug-diagnostic instructions with other SBST approaches in the literature we will utilize LEON3 processor as a benchmark for the L1 cache comparisons that has been also utilized in previous SBST approaches.

In [72] and [75] four pseudo-instructions (a cache disable, a main memory write, a cache enable and a load miss and refill) were used to apply the March write operations for the tag arrays due to the need of setting up data inconsistency. A similar pseudo-instruction sequence is utilized for March read operation. In the proposed methodology, March writes and reads can be performed in a low cost way with a single DCA instruction. Moreover in [72] and [73], pairs of instructions that complement partially in some bits are utilized to overcome the DB composition challenge of IL1-Data array. The entire algorithm should be repeated multiple times to cover all instruction bits. In [79], a more effective approach was utilized that fills the cache lines with instructions with complementary bits but also the whole test has to be repeated at least two times. Both solutions have a negative effect on test time. The utilization of DCA instructions overcomes this difficulty. Any DB can be utilized as the cache will be invalidated at the end of the test.

Finally, in [72] and [79], two different solutions were proposed to implement a descending address order in L1 instruction cache. In [72], the test execution is performed in processor's trace mode, while in [79] a reordering function modifies the access history of the cache lines. Indeed, both solutions overcome the abovementioned difficulty but they excessively increase the test time. The proposed methodology overcomes the challenge to implement a March descending address order in both L1 instruction cache and TLBs, by controlling the WS, SS and LWS fields of DCA instructions to setup a cache access descending address order.

The same reasoning is also verified by comparing the statistical results with the different SBST approaches when the proposed methodology is applied to the L1 caches of LEON3 processor. LEON3 is a publicly available processor designed by Aeroflex Gaisler, which implements a SPARC V8 compliant architecture, supports reconfigurable L1 data and instruction cache and includes DCA instruction in its ISA. For the sake of comparisons, we have configured the processor to include 2KB 2-way set associative data and instruction L1 caches in order to compare the proposed methodology with other SBST approaches in the literature. The SPARC V8 ISA includes privileged store/load instructions, denoted as alternate load/store (*lda/sta* instructions). These instructions can directly access cache arrays for diagnostic purposes by specifying alternate space identifiers (ASIs) that are defined by the SPARC architecture for both write and read access at supervisor level. These instructions have been used as DCA instructions to implement March write/read operations in a similar way that we have used *stxa/l*dxa instructions in case of OpenSPARC T1.

Table 16: Statistics comparison for D-Cache (March C-)

Processor	**DL1-Tag**			**DL1-Data**		
	Array Size	**Test size (bytes)**	**Test Time (cycles)**	**Array Size**	**Test size (bytes)**	**Test Time (cycles)**
ARM-comp. [75]	64x21	2,560	139,219	2K	686	187,453
LEON3 (*Proposed*)	2x 32x22	384	3,123	2K	384	23,729
Improvement		85%	98%		88%	87%

In Table 16, two benchmarks with similar ISA for a given March test (March C-) are compared. The experimental results in [75] for the data cache of an ARM-compatible processor are compared with our experimental results for a same size (2K) data cache of the LEON3 processor. Even though a direct numerical comparison is not feasible between different benchmarks, the methodology favors to the one that is presented in [75] since the test time is improved by 92.5% on average (for both arrays) for the same size data caches and the same March test. Finally, the test code size was improved by 86.5%.

Table 17: Statistics comparison for I-Cache (March SS)

Processor	IL1-Tag			IL1-Data		
	Array Size	Test size (bytes)	Test Time (cycles)	Array Size	Test size (bytes)	Test Time (cycles)
LEON3 [79]	2x 32x22	3,608	413,243	2K	3,944	1,922,804
LEON3 (*Proposed*)	2x 32x22	632	36,007	2K	632	288,073
Improvement		82%	91%		84%	85%

In Table 17, a direct comparison on the same benchmark between the methodology of [79] and our methodology is presented. Both methodologies apply the same March test (March SS) to the same size instruction cache of LEON3 processor. Note that the proposed methodology significantly improves both test code size and test time of March SS application to LEON3's L1 instruction cache. The total test code size for both arrays (IL1-Tag & IL1-Data) is improved by 83% on average, while the total test time is improved by 88% on average (for both arrays). This improvement is achieved by exploiting DCA instructions to effectively implement the March tests.

SBST approaches for applying March tests to TLB arrays have not been presented in the literature. Therefore, in order to demonstrate the effectiveness of the proposed SBST methodology to TLBs when DCAs are exploited, we have also implemented the same March tests for OpenSPARC T1 by utilizing the miss & refill TLB mechanism. March write, read and compare operations have been mapped to *ldx* instructions for the D-TLB arrays whereas the same instructions have been mapped to *jmpl* instructions for the I-TLB arrays. When the native TLB miss & refill mechanism is utilized, either MMU demap operation or an access reordering function (RO) should be also utilized to overcome the testability challenges. For the sake of comparison of all the available implementation choices that are presented in the methodology, we have chosen to exploit the OpenSPARC's T1 demap operation along with the miss & refill mechanism for March writes for the D-TLB arrays and the RO function [79] along with the miss & refill mechanism for March writes for I-TLB arrays. The comparison results for all the TLB arrays of the two SBST approaches (with and without the utilization of DCA instructions) are presented in Table 18 and Table 19. It is clear that when DCA instructions are exploited the test time is significantly improved. The improvement compared to the approach that utilizes miss & refill TLB mechanism is 37% for the D-TLB and 91% for the I-TLB. Note that in case of I-TLB the utilization of RO function for March write implementation instead of the demap mechanism affects seriously the test time. Finally, test code size was improved by 3% (on average for both arrays) for the D-TLB and by 35% (on average for both arrays) for I-TLB.

Table 18: Statistics comparison for D-TLB (March SS)

Processor	DTLB-Tag			DTLB-Data		
	Array Size	Test size (bytes)	Test Time (cycles)	Array Size	Test size (bytes)	Test Time (cycles)
OpenSPARC (Miss&Refill +Demap)	64x59	704	78,208	64x43	704	78,208
OpenSPARC (DCA Instr.)	64x59	684	49,240	64x43	684	49,240
Improvement		3%	37%		3%	37%

Table 19: Statistics comparison for I-TLB (March SS)

Processor	ITLB-Tag			ITLB-Data		
	Array Size	Test size (bytes)	Test Time (cycles)	Array Size	Test size (bytes)	Test Time (cycles)
OpenSPARC (Miss&Refill +RO function)	64x59	808	469,400	64x43	808	469,400
OpenSPARC (DCA Instr.)	64x59	590	51,532	64x43	460	33,944
Improvement		27%	89%		43%	93%

7. CONCLUSIONS & FUTURE RESEARCH

Modern processors are dominated by on-chip caches which occupy up to 90% of the processor silicon area and are organized in multiple levels. Typically, "lower-level" caches have smaller size, smaller blocks, and smaller associativity, while "higher-level" caches have larger size, larger blocks and higher associativity. Most of the modern processors have at least three low level small caches: an instruction L1 cache to speed up instruction fetch (L1 I-Cache), a data L1 cache to speed up data loads and stores (L1 D-Cache) and a unified Translation Lookaside Buffer (TLB) that is used as a page table cache to speed up virtual-to-physical address translation both for instructions and data. Although L1 caches and TLBs are small caches, they are extremely critical for the system performance because the system pays significant large penalties when L1 cache or TLB misses occur. Hardware defects in these low level cache arrays during normal operation may cause either erroneous cache misses that degrade the system's performance, or unpredicted system behaviour. Thus, high quality on-line tests that target small caches are essential for modern processors.

Despite the wide use of MBIST in on-chip memories, small memory arrays with typical sizes in the order of Kbytes such as L1 caches, TLBs, register files, FIFOs etc. may not justify the cost of adding programmable MBIST schemes because of its impact on chip area and performance. SBST has increased flexibility to apply March tests and can successfully deal with the challenges of on-line testing of such small memory arrays that lack MBIST hardware.

The contribution of this thesis is to introduce an SBST program development methodology for low cost on-line fault detection of processor small caches (L1 caches and TLBs) based on state-of-the-art memory March test algorithms. The methodology overcomes the testability challenges that are due to the implicit access of L1 cache and TLB arrays and realizes March write, read and compare (in TLB tag arrays only) operations by leveraging existing special purpose instructions that modern ISAs implement for debug-diagnostic purposes. Moreover, it exploits performance monitoring hardware and the processor's trap handler mechanism. Finally, a multithreaded optimization of the proposed methodology, that elaborates the low level multiple sub-bank organization of modern cache designs to exploit the thread level parallelism of chip multithreaded, multicore architectures in order to speedup SBST March test execution while preserving the March test quality, was also presented.

The proposed methodology and its multithreaded optimization were applied to the L1 caches and TLBs of three processor benchmarks: a) OpenRISC 1200, b) LEON3 and c) OpenSPARC T1 to demonstrate the effectiveness of the proposed methodology, its high adaptability and the significant improvements in terms of SBST March test execution time when applied.

Experimental results on the L1 cache arrays of LEON3 and the TLB arrays of OpenSPARC T1 show a significant improvement in terms of test time (86% for instruction L1 cache, 87% for the data L1 cache, about 40% for D-TLB and about 82% for I-TLB) and test code size (83% for instruction L1 cache, 86% for the data L1 cache, 3% for D-TLB and 35% for I-TLB) when the methodology is applied to the same benchmarks (LEON3 for L1 caches and OpenSPARC T1 for TLBs) and such DCA instructions are exploited compared to SBST solutions that don't utilize these types of instructions. Moreover, experimental results show a speedup of more than 1.7 (for two threads) and more than 3.7 (for four threads) in test time when the proposed multithreaded optimization is applied for the L1 caches of OpenSPARC T1.

Finally, a test evaluation framework was implemented in this thesis for several on-line periodic test scenarios in order to evaluate the system performance overhead of the proposed methodology. Simulation results show a performance overhead of less than 11% in strict periodic test scenarios and less than 6% in regular periodic test scenarios (e.g. testing all core L1 caches every one minute) for multicore architectures.

Possible future research topics that have not been covered by this thesis and can further extend the proposed methodology are listed below:

- *Extending the proposed methodology to apply March tests to L2 caches*: L2 caches are large memory arrays with sizes in the order of Mbytes. Therefore, SBST March test implementation for such large arrays can be challenging since the test execution time is significantly larger. Moreover, L2 caches are usually organized as set associative caches with pseudorandom replacement police that makes SBST March test implementation more challenging.
- *Extending the proposed methodology to apply March tests to embedded cache arrays with low power constraints:* Energy requirements were only recently added to the list of parameters defining the cost of SBST routines, together with more traditional metrics like test program size and test execution time, especially for on-line testing of microprocessors integrated in mobile devices, in order to maximize battery life and to avoid long term reliability problems. Microprocessors in such

mobile devices also include embedded cache arrays (at least small size L1 caches are present in all embedded processors). Therefore, the optimization of the implemented software routines to apply low power March tests to embedded cache arrays will further extend the importance of the proposed SBST methodology.

- *Extending the proposed methodology to apply March tests to other SoC embedded memories* (Scratchpad memories e.t.c): SoC systems are dominated by a large number of small and big embedded memories that are utilized for several purposes in the chip. Extending the proposed methodology to apply SBST March test to these off-processor embedded memories is challenging since these memories have various levels of visibility to the processor’s ISA based on their functionality and their position in the interconnecting network of the SoC system.

ACRONYMS

AC	Alternating Current
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
BCM	Bus Connected Memories
BIST	Built-In Self-Test
CAM	Content Addressable Memory
CMOS	Complementary Metal Oxide Semiconductor
CUT	Circuit Under Test
DC	Direct Current
DCA	Direct Cache Access
DFT	Design For Testability
ECC	Error Correction Code
FSM	Finite State Machine
IP	Intellectual Property
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
LFSR	Linear Feedback Shift Register
LRU	Least Recently Used
MBIST	Memory Built-In Self-Test
MIPS	Microprocessor without Interlocked Pipeline Stages
PI	Primary Input
PO	Primary Output
RAM	Random Access Memory
RIC	Resistance-Inductance-Capacitance
RISC	Reduced Instruction Set Computers
RTL	Register Transfer Level
RTPG	Random Test Pattern Generation
SBST	Software-Based Self-Test
SOC	System On-Chip
SRAM	Static Random Access Memory
TLB	Translation Lookaside Buffer
TPG	Test Pattern Generation
VLSI	Very Large Scale Integration Systems

APPENDIX I – FAULT COVERAGE STATISTICS

In order to evaluate the effectiveness of the self-test routines we have used RAMSES memory fault simulator [88]. RAMSES consists of a simulation engine and numerous fault descriptors. Fault coverage is determined by evaluating the fault descriptors for predefined conditions. When coupling faults are concerned, the rest of the array cells except from the aggressor cell are possible victim cells. We have extended RAMSES to include fault descriptors on the basis of FPs [11] to include a) all the unlinked static storage faults [11] and b) CAM comparison faults [17] and we have implemented a test framework to bridge cache traces of Mentor Graphics' ModelSim and Synopsys VCS simulator with RAMSES to fault grade the cache arrays. RAMSES fault simulator exhaustively injected faults in every cell for all unlinked static faults. We have fault graded the embedded cache arrays (both data and tag array) for all L1 caches that are included in our three benchmarks for all static unlinked single-cell storage faults and 2-cell coupling storage faults. The considered functional fault models for the considered storage faults are listed below, and the corresponding fault primitives that show the faults behavior are described in Chapter 2.

Single Cell Faults

- State Faults (SF)
- Transition Faults (TF)
- Write Destructive Faults (WDF)
- Read Destructive Faults (RDF)
- Deceptive Read Destructive Faults (DRDF)
- Incorrect Read Faults (IRF)

Cell Coupling Faults (2-cell)

- Coupling State Faults (CFst),
- Coupling Disturb Faults (CFds)
- Coupling Transition Faults (CFtr)
- Coupling Write Destructive Faults (CFwd)
- Coupling Read Destructive Faults (CFrd)
- Coupling Deceptive Read Destructive Faults (CFdr)
- Coupling Incorrect Read Faults (CFir)

For OpenSPARC's TLBs, that include CAM memories in tag arrays we have fault graded the tag arrays with RAMSES fault simulator both for the abovementioned storage faults and the comparison faults that are listed below:

Comparison faults

- Stuck-Match Faults (SMF)
- Stuck-MisMatch Faults (SMMF)
- Conditional-Match Faults (CMF)
- Partially-Match Faults (PMF)
- Equivalence-MisMatch Faults (EMMF)
- Inequivalence-Match Faults (IMF)
- Cross-Match Faults (XMF)
- Cross-Mismatch Faults (XMMF)

The achieved fault coverage for every SBST routine that was implemented by the proposed methodology that exploits DCA instructions for every embedded cache arrays that has been fault graded is listed below in Tables 20-53.

LEON3 – L1 Caches

Table 20: LEON3 DL1–Tag SRAM array, Size: 64x29 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 21: LEON3 DL1–Tag SRAM array, Size: 64x29 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 22: LEON3 DL1–Data SRAM array, Size: 512x32 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 23: LEON3 DL1–Data SRAM array, Size: 512x32 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 24: LEON3 IL1–Tag SRAM array, Size: 64x29 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 25: LEON3 IL1–Tag SRAM array, Size: 64x29 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 26: LEON3 IL1–Data SRAM array, Size: 512x32 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 27: LEON3 IL1–Data SRAM array, Size: 512x32 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

OpenRISC 1200 – L1 Caches

Table 28: OpenRISC 1200 DL1–Tag SRAM array, Size: 256x20, 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 29: OpenRISC 1200 DL1–Tag SRAM array, Size: 256x20, 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 30: OpenRISC 1200 DL1–Data SRAM array, Size: 1024x32, 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 31: OpenRISC 1200 DL1–Data SRAM array, Size: 1024x32, 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 32: OpenRISC 1200 IL1–Tag SRAM array, Size: 256x20, 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	99%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 33: OpenRISC 1200 IL1–Tag SRAM array, Size: 256x20, 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	99%	-	99%	-	99%	-
March U	-	99%	-	99%	-	99%	-
March MSS	99%	99%	99%	99%	99%	99%	99%
March SS	99%	99%	99%	99%	99%	99%	99%

Table 34: OpenRISC 1200 IL1–Data SRAM array, Size: 1024x32, 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	99%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 35: OpenRISC 1200 IL1–Data SRAM array, Size: 1024x32, 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	99%	-	99%	-	99%	-
March U	-	99%	-	99%	-	99%	-
March MSS	99%	99%	99%	99%	99%	99%	99%
March SS	99%	99%	99%	99%	99%	99%	99%

OpenSPARC T1 – L1 Caches

Table 36: OpenSPARC T1 DL1–Tag SRAM array, Size: 64x132 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 37: OpenSPARC T1 DL1–Tag SRAM array, Size: 64x132 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 38: OpenSPARC T1 DL1–Data SRAM array, Size: 128x288 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 39: OpenSPARC T1 DL1–Data SRAM array, Size: 128x288 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 40: OpenSPARC T1 IL1–Tag SRAM array, Size: 64x132 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 41: OpenSPARC T1 IL1–Tag SRAM array, Size: 64x132 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 42: OpenSPARC T1 IL1–Data SRAM array, Size: 128x272 (x4 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 43: OpenSPARC T1 DL1–Data SRAM array, Size: 128x272 (x4 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

OpenSPARC T1 – TLBs

Table 44: OpenSPARC T1 DTLB–Tag CAM array, Size: 64x132 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 45: OpenSPARC T1 DTLB –Tag CAM array, Size: 64x132 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 46: OpenSPARC T1 DTLB –Tag CAM array, Size: 64x132 (x2 banks), Comparison faults

March test	Fault Coverage							
	SMF	SMMF	CMF	PMF	EMMF	IMF	XMF	XMMF
March of [89]	92%	93%	92%	91%	91%	92%	88%	92%
March CFT [90]	92%	93%	92%	91%	91%	92%	88%	92%

Table 47: OpenSPARC T1 DTLB–Data SRAM array, Size: 128x288 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 48: OpenSPARC T1 DTLB–Data SRAM array, Size: 128x288 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 49: OpenSPARC T1 ITLB–Tag CAM array, Size: 64x132 (x2 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 50: OpenSPARC T1 ITLB –Tag CAM array, Size: 64x132 (x2 banks), 2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 51: OpenSPARC T1 ITLB –Tag CAM array, Size: 64x132 (x2 banks), Comparison faults

March test	Fault Coverage							
	SMF	SMMF	CMF	PMF	EMMF	IMF	XMF	XMMF
March of [89]	92%	93%	92%	91%	91%	92%	88%	92%
March CFT [90]	92%	93%	92%	91%	91%	92%	88%	92%

Table 52: OpenSPARC T1 ITLB–Data SRAM array, Size: 128x272 (x4 banks), 1-cell single faults

March test	Fault Coverage						
	SA	TF	WDF	RDF	DRDF	IRF	AF
March C-	100%	100%	-	100%	-	100%	100%
March U	100%	100%	-	100%	-	100%	100%
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

Table 53: OpenSPARC T1 ITLB–Data SRAM array, Size: 128x272 (x4 banks),
2-cell coupling faults

March test	Fault Coverage						
	CFst	CFds	CFtr	CFwd	CFrd	CFdrd	CFir
March C-	-	100%	-	100%	-	100%	-
March U	-	100%	-	100%	-	100%	-
March MSS	100%	100%	100%	100%	100%	100%	100%
March SS	100%	100%	100%	100%	100%	100%	100%

APPENDIX II – SBST ROUTINES CODE SNIPPETS

LEON3 – IL1-Data March SS

```
#include "testmod.h"
#include "leon3.h"
marchtest()
{
report_subtest(MARCH_TEST);   //Icache data - MARCH MSS
                                            //Icache size 2-way set assosiative (2*2kbytes)
                                            //SRAM size:  512 x 32 x 2(ways)
asm(
"    .text           \n"
"    .align 4      \n"
"test:              \n"
"    nop            \n"
"    nop             \n"

"    add %r0,0xfff, %r1  \n"
"    add %r0,0xfff, %r2  \n"
"    add %r0,0xfff, %r3  \n"
"    add %r0,0xfff, %r4  \n"
"    add %r0,0xfff, %r5  \n"          //way 0 load reg
"    add %r0,0xfff, %r6  \n"          //way 1 load reg
"    add %r0,0xfff, %r7  \n"          //final result - golden model

"    lda [%r0] 0x2, %r1      \n"  //Disable ICACHE during test
"    and  %r1,0xfffffffc,%r1          \n"
"    sta  %r1, [%r0] 0x2     \n"

"    set 0x00000000,%r1 \n"       // db
"    set 0xffffffff,%r2 \n"     // db_inv
"    set 0,          %r3 \n"     // way 0 start
"    set 2048,      %r4 \n"   // way 1 start
"    set 0,          %r5 \n"     //way 0 load reg
"    set 0,         %r6 \n"     //way 1 load reg
"    set 0,         %r7 \n"     //final result - golden model
"    set 0xfffffBff,%r8 \n"             // db_inv

"1:  sta %r1, [%r3] 0xd     \n"  //w0 >
"    sta %r1, [%r4] 0xd      \n"
"    add  %r3,  4, %r3       \n"
"    add  %r4,  4, %r4       \n"
"    cmp  %r3, 2048          \n"
"    bne  1b                 \n"
"    nop                     \n"

"    set 0,          %r3         \n"
"    set 2048,      %r4         \n"

"2:  lda [%r3] 0xd, %r5     \n"  //r0,r0,w1,w1 >
"    xor %r1, %r5,  %r5     \n"
"    add %r7, %r5,  %r7    \n"
"    lda [%r3] 0xd, %r5      \n"
"    xor %r1, %r5,  %r5     \n"
"    add %r7, %r5,  %r7    \n"
"    sta %r2, [%r3] 0xd      \n"
"    sta %r2, [%r3] 0xd      \n"

"    lda [%r4] 0xd, %r6      \n"
"    xor %r1, %r6,  %r6     \n"
"    add %r7, %r6,  %r7    \n"
"    lda [%r4] 0xd, %r6      \n"
"    xor %r1, %r6,  %r6      \n"
"    add %r7, %r6,  %r7      \n"
"    sta %r2, [%r4] 0xd      \n"
"    sta %r2, [%r4] 0xd      \n"

"    add  %r3,  4, %r3       \n"
"    add  %r4,  4, %r4       \n"
"    cmp  %r3, 2048          \n"
"    bne  2b                 \n"
"    nop                     \n"

"    set 0,          %r3     \n"
"    set 2048,       %r4     \n"

"3:  lda [%r3] 0xd, %r5      \n"  //r1,r1,w0,w0 >
"    xor %r2, %r5,  %r5      \n"
"    add %r7, %r5,  %r7      \n"
"    lda [%r3] 0xd, %r5      \n"
"    xor %r2, %r5,  %r5      \n"
"    add %r7, %r5,  %r7      \n"
"    sta %r1, [%r3] 0xd      \n"
"    sta %r1, [%r3] 0xd      \n"

"    lda [%r4] 0xd, %r6      \n"
"    xor %r2, %r6,  %r6      \n"
"    add %r7, %r6,  %r7      \n"
"    lda [%r4] 0xd, %r6      \n"
"    xor %r2, %r6,  %r6      \n"
"    add %r7, %r6,  %r7      \n"
"    sta %r1, [%r4] 0xd      \n"
"    sta %r1, [%r4] 0xd      \n"

"    add  %r3, 4, %r3        \n"
"    add  %r4, 4, %r4        \n"
"    cmp  %r3, 2048          \n"
"    bne  3b                 \n"
"    nop                     \n"

"    set 2044,               %r3         \n"
"    set 4092,       %r4     \n"

"4:  lda [%r3] 0xd, %r5      \n"  //r0,r0,w1,w1 <
"    xor %r1, %r5,  %r5      \n"
"    add %r7, %r5,  %r7      \n"
"    lda [%r3] 0xd, %r5      \n"
"    xor %r1, %r5,  %r5      \n"
"    add %r7, %r5,  %r7      \n"
"    sta %r2, [%r3] 0xd      \n"
"    sta %r2, [%r3] 0xd      \n"

"    lda [%r4] 0xd, %r6      \n"
"    xor %r1, %r6,  %r6      \n"
"    add %r7, %r6,  %r7      \n"
"    lda [%r4] 0xd, %r6      \n"
"    xor %r1, %r6,  %r6      \n"
"    add %r7, %r6,  %r7      \n"
"    sta %r2, [%r4] 0xd      \n"
"    sta %r2, [%r4] 0xd      \n"

"    add  %r3, -4, %r3       \n"
"    add  %r4, -4, %r4       \n"
"    cmp  %r3, -4                        \n"
"   bne  4b                    \n"
"   nop                        \n"

"   set 2044,            %r3        \n"
"   set 4092,      %r4     \n"

"5:  lda [%r3] 0xd, %r5     \n"  //r1,r1,w0,w0 <
"   xor %r2, %r5,  %r5     \n"
"   add %r7, %r5,  %r7     \n"
"   lda [%r3] 0xd, %r5     \n"
"   xor %r2, %r5,  %r5     \n"
"   add %r7, %r5,  %r7     \n"
"   sta %r1, [%r3] 0xd     \n"
"   sta %r1, [%r3] 0xd     \n"

"   lda [%r4] 0xd, %r6     \n"
"   xor %r2, %r6,  %r6     \n"
"   add %r7, %r6,  %r7     \n"
"   lda [%r4] 0xd, %r6     \n"
"   xor %r2, %r6,  %r6     \n"
"   add %r7, %r6,  %r7     \n"
"   sta %r1, [%r4] 0xd     \n"
"   sta %r1, [%r4] 0xd     \n"

"   add  %r3, -4, %r3      \n"
"   add  %r4, -4, %r4      \n"
"   cmp  %r3, -4           \n"
"   bne  5b                \n"
"   nop                    \n"

"   set 0,        %r3      \n"
"   set 2048,     %r4      \n"

"6:  lda [%r3] 0xd, %r5     \n"  //r0 >
"   lda [%r4] 0xd, %r6     \n"
"   xor %r1, %r5,  %r5     \n"
"   add %r7, %r5,  %r7     \n"
"   xor %r1, %r6,  %r6     \n"
"   add %r7, %r6,  %r7     \n"
"   add  %r3,  4, %r3      \n"
"   add  %r4,  4, %r4      \n"
"   cmp  %r3, 2048         \n"
"   bne  6b                \n"
"   nop                    \n"



"   lda [%r0] 0x2, %r1     \n"  //Enable ICACHE after test
"   xor  %r1,0x3,  %r1     \n"
"   sta  %r1, [%r0] 0x2    \n"

"   nop          \n"
"   nop           \n"
"   nop          \n"
"   nop           \n"
"   nop          \n"
"   nop           \n"
);

   return(0);}
```

OpenRISC 1200 – DL1-Tag March SS

```
############################################################
# AUTHOR: gthe
# FILE: DL1-Tag, Size of Cache 4k, Size of tag 256x20
# VERSION: March SS (22n) all static linked faults!!!
#############################################################

/* Basic instruction set test */
#include "../support/spr_defs.h"
#include "../support/board.h"

.global _main
.global _buserr_except
.global _dpf_except
.global _ipf_except
.global _lpint_except
.global _align_except
.global _illegal_except
.global _hpint_except
.global _dtlbmiss_except
.global _itlbmiss_except
.global _range_except
.global _syscall_except
.global _res1_except
.global _trap_except
.global _res2_except

        .section .stack
        .space 0x1000
_tmp_stack:

  .section .text

_buserr_except:
_dpf_except:
_ipf_except:
_lpint_except:
_align_except:
_illegal_except:
_hpint_except:
_dtlbmiss_except:
_itlbmiss_except:
_range_except:
_syscall_except:
_res1_except:
_trap_except:
_res2_except:

_main:

################################
#Applying March C- algorithm to 256x21 data cache tag
#
#Steps:
# 1. Write 0 to all cells
# 2. From 0 to 255 read 0, read 0, write 0, read 0, write 1
# 3. From 0 to 255 read 1, read 1, write 1, read 1, write 0
# 4. From 255 to 0 read 0, read 0, write 0, read 0, write 1
# 5. From 255 to 0 read 1, read 1, write 1, read 1, write 0
# 6. Read 0 from all cells
# Write 0 instruction l.mtspr r0,r1,SPR_DCBPR
# Write 1 instruction l.mtspr r0,r30,SPR_DCBPR
# Read  instruction l.addi  r5,r1,0x0 sta ports tou tag
# Read detection MONO mesw cache hit sthn antistoixi dieythinsi!
#################################


l.add r30,r0,r0
l.add r1,r0,r0


l.movhi r4,0xffff
l.ori r4,r4,0xf000

l.movhi r11,0x0000
l.ori   r11,r11,0x81

l.movhi r12,0x0000
l.ori   r12,r12,0x01

l.mtspr r0,r11,SPR_PCMR(0)

mats_loop_1:

l.mtspr r0,r1,SPR_DCBPR

l.addi  r1,r1,16
l.sfeqi  r30,4080
l.bnf   mats_loop_1
l.addi  r30,r30,16

l.add r30,r0,r0
l.add r1,r0,r0

mats_loop_2:

l.nop
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)
l.nop
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.mtspr r0,r1,SPR_DCBPR
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.mtspr r0,r30,SPR_DCBPR
l.nop

l.addi  r1,r1,16
l.sfeqi  r30,4080
l.bnf   mats_loop_2
l.addi  r30,r30,16

l.add r30,r0,r0
l.add r1,r0,r0

l.nop


mats_loop_3:
```

```
l.nop
l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)

l.nop
l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)

l.mtspr r0,r30,SPR_DCBPR

l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)

l.mtspr r0,r1,SPR_DCBPR
l.nop

l.addi  r1,r1,16
l.sfeqi  r30,4080
l.bnf   mats_loop_3
l.addi  r30,r30,16

l.addi r30,r0,4080
l.addi r1,r0,4080

mats_loop_4:

l.nop
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.nop
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.mtspr r0,r1,SPR_DCBPR
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.mtspr r0,r30,SPR_DCBPR
l.nop

l.addi  r1,r1,-16
l.sfeqi  r30,0
l.bnf   mats_loop_4
l.addi  r30,r30,-16

l.addi r30,r0,4080
l.addi r1,r0,4080

mats_loop_5:

l.nop
l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)

l.nop
l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)
```

```
l.mtspr r0,r30,SPR_DCBPR

l.addi  r5,r1,0x7000
l.add r6,r4,r1
l.lwz r5,0(r6)

l.mtspr r0,r1,SPR_DCBPR
l.nop

l.addi  r1,r1,-16
l.sfeqi  r30,0
l.bnf   mats_loop_5
l.addi  r30,r30,-16

l.add r30,r0,r0
l.add r1,r0,r0

mats_loop_6:

l.nop
l.addi  r5,r1,0x7000
l.lwz r5,0(r1)

l.addi  r1,r1,16
l.sfeqi  r30,4080
l.bnf   mats_loop_6
l.addi  r30,r30,16

l.add r30,r0,r0
l.add r1,r0,r0

l.mtspr r0,r12,SPR_PCMR(0)
```

OpenSPARC T1 – IL1-Tag March SS

```
SECTION .SBST Text_VA=SBST_TEXT_ADDR_VA
attr_text {
       Name = .SBST,
         VA= SBST_TEXT_ADDR_VA,
         hypervisor
       }

.text
.global sbst
.global Verify_pic

sbst:

//Initialize a performance counter to monitor instruction cache misses
      set 0x21, %l0     // Turn OFF PIC
      set 0x23, %l1   // Turn ON  PIC

/*********************************************************
       %g1 = instruction to be written (retl)
       %g2 = cache tag address
       %g3 = way counting (0...3)
       %g4 = tag address step in every iteration
       %g5 = tag to be written (all-1 )
       %g6 = tag to be written (all-0 )
       %g7 = Last cache address step
      %l0 = turn PIC OFF
      %l1 = turn PIC ON
      %l2 = VA address step in every iteration
      %l3 = VA address OFFSET for every way change
      %l4 = tag offset for way change
      %l5 = PIC read in every Read & Verify
      %l6 = VA for tag access through CALL
      %l7 = address OFFSET for way change
*******************************************************/

//MARCH SS w0 up

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                    //OFFSET to change set in icache
        set  0x1111111, %l4                         //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db1, %l2, %l6

all_sets1:
         setx 0x0000000000000000, %o0, %g2
         setx 0x0000000400000000 ,%o0, %g6       // 7 lsb hex bits define the desired tag and bit (34)
defines the valid bit
         setx db1, %l2, %l6
         add  %g0, %g0,  %g3
         add  %g0, %g4,  %g2
         add  %l6, %l2,  %l6

way_write1:

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g6, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi

        add %g2, %l7, %g2           !! switch way of cache
        add %g6, %l4, %g6           !! switch tag
        add %l6, %l3, %l6    !! switch VA to access a different way
        add %g3, 0x1, %g3           !! repeat the way writing 4 times (4-way cache)
        cmp %g3, 0x4
        bl %xcc, way_write1
        nop
        nop
        nop

        add %l2, 0x20, %l2
        add %g4, 0x40, %g4          !! Go to next address for all 4 ways
        cmp %g4, %g7
        bl %xcc, all_sets1
        nop
        nop

//MARCH SS r0r0w0r0w1 up

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                       //OFFSET to change set in icache
        set  0x1111111, %l4                             //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db1, %o0, %l6

all_sets2:
        setx 0x0000000000000000, %o0, %g2
        setx 0x000000040fffffff ,%o0, %g5
        setx 0x0000000400000000 ,%o0, %g6
        setx db1, %o0, %l6
        add %g0, %g0,  %g3
        add %g0, %g4,  %g2
        add %l6, %l2,  %l6

way_write2:

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g6, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi


        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic


        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g5, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi

        add %g2, %l7, %g2           !! switch way of cache
        sub %g5, %l4, %g5           !! switch tag
        add %g6, %l4, %g6           !! switch tag
        add %l6, %l3, %l6    !! switch VA to access a different way
        add %g3, 0x1, %g3            !! repeat the way writing 4 times (4-way cache)
        cmp %g3, 0x4
        bl %xcc, way_write2
        nop
        nop
        nop

        add %l2, 0x20, %l2
        add %g4, 0x40, %g4           !! Go to next address for all 4 ways
        cmp %g4, %g7
        bl %xcc, all_sets2
        nop
        nop

//MARCH SS r1r1w1r1w0 up

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                 //OFFSET to change set in icache
        set  0x1111111, %l4                      //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db1, %l2, %l6

all_sets3:
         setx 0x0000000000000000, %o0, %g2
         setx 0x000000040fffffff ,%o0, %g5
         setx 0x0000000400000000 ,%o0, %g6
         setx db_inv1, %o0, %l6
         add %g0, %g0,  %g3
         add %g2, %g4,  %g2
         add %l6, %l2,  %l6


way_write3:

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g5, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi
        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g6, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi

        add %g2, %l7, %g2           !! switch way of cache
        add %g6, %l4, %g6           !! switch tag
        sub %g5, %l4, %g5
        sub %l6, %l3, %l6    !! switch VA to access a different way
        add %g3, 0x1, %g3           !! repeat the way writing 4 times (4-way cache)
        cmp %g3, 0x4
        bl %xcc, way_write3
        nop
        nop
        nop

        add %l2, 0x20, %l2
        add %g4, 0x40, %g4          !! Go to next address for all 4 ways
        cmp %g4, %g7
        bl %xcc, all_sets3
        nop
        nop

//MARCH SS r0r0w0r0w1 down

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                   //OFFSET to change set in icache
        set  0x1111111, %l4                         //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db1, %o0, %l6
        add  %l6, 0xfe0, %l6

all_sets4:
         setx 0x0000000000001fc0, %o0, %g2
         setx 0x000000040fffffff ,%o0, %g5     // 7 lsb hex bits define the desired tag and bit (34) defines
the valid bit
         setx 0x0000000400000000 ,%o0, %g6
         setx db1, %o0, %l6
         add  %l6, 0xfe0, %l6
         add  %g0, %g0,  %g3
         sub  %g2, %g4,  %g2
         sub  %l6, %l2,  %l6

way_write4:

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic
        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g6, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        wr %g0, ASI_ICACHE_TAG, %asi
        stxa %g5, [%g2] %asi
        nop

        wr   %g0, ASI_ICACHE_INST, %asi
        stxa %g1, [%g2] %asi
        stxa %g1, [%g2+8] %asi

        add %g2, %l7, %g2           !! switch way of cache
        sub %g5, %l4, %g5           !! switch tag
        add %g6, %l4, %g6           !! switch tag
        add %l6, %l3, %l6    !! switch VA to access a different way
        add %g3, 0x1, %g3           !! repeat the way writing 4 times (4-way cache)
        cmp %g3, 0x4
        bl %xcc, way_write4
        nop

        add %l2, 0x20, %l2
        add %g4, 0x40, %g4          !! Go to next address for all 4 ways
        cmp %g4, %g7
        bl %xcc, all_sets4
        nop

//MARCH SS r1r1w1r1w0 down

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                   //OFFSET to change set in icache
        set  0x1111111, %l4                         //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db_inv1, %o0, %l6
        add  %l6, 0xfe0, %l6

all_sets5:
         setx 0x0000000000001fc0, %o0, %g2
         setx 0x000000040fffffff ,%o0, %g5
```

```
          setx 0x0000000400000000 ,%o0, %g6
          setx db_inv1, %o0, %l6
          add  %l6, 0xfe0, %l6
          add  %g0, %g0,  %g3
          sub  %g2, %g4,  %g2
          sub  %l6, %l2,  %l6

way_write5:

         wr %l1, 0 , %pcr
         call %l6
         nop
         wr %l0, 0 , %pcr
         rd %pic, %l5
         add %l5,-4,%l5
         wr  %l5, 0 , %pic

         wr %l1, 0 , %pcr
         call %l6
         nop
         wr %l0, 0 , %pcr
         rd %pic, %l5
         add %l5,-4,%l5
         wr  %l5, 0 , %pic

         wr %g0, ASI_ICACHE_TAG, %asi
         stxa %g5, [%g2] %asi
         nop

         wr   %g0, ASI_ICACHE_INST, %asi
         stxa %g1, [%g2] %asi
         stxa %g1, [%g2+8] %asi

         wr %l1, 0 , %pcr
         call %l6
         nop
         wr %l0, 0 , %pcr
         rd %pic, %l5
         add %l5,-4,%l5
         wr  %l5, 0 , %pic

         wr %g0, ASI_ICACHE_TAG, %asi
         stxa %g6, [%g2] %asi
         nop

         wr   %g0, ASI_ICACHE_INST, %asi
         stxa %g1, [%g2] %asi
         stxa %g1, [%g2+8] %asi

         add %g2, %l7, %g2           !! switch way of cache
         add %g6, %l4, %g6           !! switch tag
         sub %g5, %l4, %g5           !! switch tag
         sub %l6, %l3, %l6     !! switch VA to access a different way
         add %g3, 0x1, %g3           !! repeat the way writing 4 times (4-way cache)
         cmp %g3, 0x4
         bl %xcc, way_write5
         nop
         nop
         add %l2, 0x20, %l2
         add %g4, 0x40, %g4          !! Go to next address for all 4 ways
         cmp %g4, %g7
         bl %xcc, all_sets5
         nop
```

```
        nop

//MARCH SS r0 down

        set  0x81c3e008, %g1
        set  0x0, %g4
        set  0x0, %l2
        set  0x2000, %g7
        set  0x10000, %l7                    //OFFSET to change set in icache
        set  0x1111111, %l4                          //different tag offset
        setx 0x0000111111111000, %o0, %l3
        setx db1, %o0, %l6
        add  %l6, 0xfe0, %l6

all_sets6:
         setx 0x0000000000001fc0, %o0, %g2
         setx 0x000000040fffffff ,%o0, %g5
         setx db1, %o0, %l6
         add  %l6, 0xfe0, %l6
         add  %g0, %g0,  %g3
         sub  %g2, %g4,  %g2
         sub  %l6, %l2,  %l6

way_write6:

        wr %l1, 0 , %pcr
        call %l6
        nop
        wr %l0, 0 , %pcr
        rd %pic, %l5
        add %l5,-4,%l5
        wr  %l5, 0 , %pic

        add %g2, %l7, %g2           !! switch way of cache
        sub %g5, %l4, %g5           !! switch tag
        add %l6, %l3, %l6           !! switch VA to access a different way
        add %g3, 0x1, %g3           !! repeat the way writing 4 times (4-way cache)
        cmp %g3, 0x4
        bl %xcc, way_write6
        nop
        nop
        nop

        add %l2, 0x20, %l2
        add %g4, 0x40, %g4          !! Go to next address for all 4 ways
        cmp %g4, %g7
        bl %xcc, all_sets6
        nop
Verify_pic:

        rd %pic, %g1
        and %g1, 0x0fff, %g1
        cmp %g0, %g1                         //check if pic.l is zero
        bne diag_fail
        nop
```

REFERENCES

[1] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, Vol.1, no.1, pp. 11- 33, Jan.-March 2004

[2] International Technology Roadmap for Semiconductors, Executive Summary, 2011 Edition, http://www.itrs.net

[3] M.L. Bushnell and V.D. Agrawal, “Essentials of Electronic Testing”, Kluwer Academic Publishers, MA, USA, 2000.

[4] A.K. Stevens, Introduction to Component Testing, Reading, Massachusetts, Addison-Wesley, 1986

[5] F. Jensen and N.E. Peterson, Burn-in, Chichester, John Wiley h Sons, Inc., UK., 1982.

[6] Y. Min, “Why RTL ATPG? “,Journal of Computer Science and Technology, Vol. 17, no. 2, pp. 113-117, 2002.

[7] H. Al-Assad, B. T. Murray, and J. P. Hayes, “Online BIST for embedded systems”, IEEE Design & Test of Computers, Vol. 15, no. 4, pp. 17–24, 1998.

[8] M. Nicolaidis and Y. Zorian, “On-line testing for VLSI—a compendium of approaches”, Journal of Electronic Testing: Theory Applications, Vol. 12, no. 1–2, pp. 7–20, 1998.

[9] N. Oh and E. J. McCluskey, “Error detection by selective procedure call duplication for low energy consumption,” IEEE Transactions on Reliability, Vol. 51, no. 4, pp. 392–402, Dec. 2002.

[10] P. J. Tan, T. Le; K.H. Ng, P. Mantri, J. Westfall, "Testing of UltraSPARC T1 Microprocessor and its Challenges", in Proc. of IEEE International Test Conference (ITC), 2006, paper 16.1.

[11] S. Hamdioui et al, “March SS: A Test for All Static Simple RAM Faults”, in Proc. of IEEE Int’l Workshop Memory Technology, Design and Testing (MTDT), 2002, pp. 95-100.

[12] Z. Al-Ars, S. Hamdioui, G. Gaydadjiev, S. Vassiliadis, "Test Set Development for Cache Memory in Modern Microprocessors", IEEE Transactions on VLSI, Vol.16, no.6, June 2008, pp.725-732.

[13] Jin-Fu Li, "Transparent-Test Methodologies for Random Access Memories Without/With ECC,", IEEE Transactions on CAD, Vol.26, no.10, pp.1888-1893, Oct. 2007.

[14] S. Boutobza, M. Nicolaidis, K.M. Lamara, A. Costa, "Programmable memory BIST", in Proc. of IEEE International Test Conference (ITC), 2005, paper 45.2.

[15] B.H. Fang, N. Nicolici, "Power-constrained embedded memory BIST architecture," in Proc. of Defect and Fault Tolerance in VLSI Systems (DFT), 2003, pp. 451- 458.

[16] D.E. Ross, T. Wood, G. Giles, "Conversion of small functional test sets of nonscan blocks to scan patterns", in Proc. of IEEE International Test Conference (ITC), 2000, pp.691-700.

[17] K.J. Lin, C.W. Wu, "Testing content-addressable memories using functional fault models and march-like algorithms", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.19, no.5, pp.577-588, May 2000

[18] A. J. van de Goor, Testing Semiconductor Memories: Theory and Practice, John Wiley & Sons, Chichester, U.K. 1991.

[19] S. Mourad and Y. Zorian, “Principles of Testing Electronic Systems“, John Wiley & Sons, Somerset, NJ, 2000.

[20] K. Sasaki, et al., “A 15ns, 1Mbit CMOS SRAM”, IEEE Journal of Solid State Circuits, Vol. 23, No. 5, pp. 1067-1072, 1988

[21] A.J. van de Goor and Z. Al-Ars, “Functional Fault Models: A Formal Notation and Taxonomy”, in Proc. of the IEEE VLSI Test Symposium (VTS), pp. 281-289, 2000

[22] C.A. Papachristou and N.B. Saghal, “An Improved Method for Detecting Functional Faults in Random Access Memories”, IEEE Transaction on Computers, Vol.31, no. 3, pp. 110-116, 1985.

[23] D.S. Suk and S.M. Reddy, “A March Test for Functional Faults in Semiconductors Random-Access Memories”, IEEE Transactions on Computers, Vol. 30, no. 12, pp. 982-985, 1981

[24] J. Zhao, S. Irrinki, M. Puri, F. Lombardi, "Testing SRAM-based content addressable memories", IEEE Transactions on Computers, Vol. 49, no. 10, pp. 1054-1063, Oct. 2000.

[25] K.J. Lin, C.W. Wu, "Testing content-addressable memories using functional fault models and march-like algorithms", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.19, no.5, pp.577-588, May 2000

[26] K. Zarrineh and S. J. Upadhyaya, “On Programmable Memory Built-In Self-Test Architectures”, in Proc. of the Design, Automation and Test in Europe Conference (DATE), pp. 708-713, 1999

[27] W. L. Wang, K. J. Lee, and J. F. Wang, “An On-Chip March Pattern Generator for Testing Embedded Memory Cores”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, no.5, pp. 730-735, October 2001

[28] P1500 SECT Task Forces. IEEE P1500 Web Site. http://grouper.ieee.org/groups/1500.

[29] S. Koranne, C. Wouters, T. Waayers, S. Kumar, R. Beurze, and G. S. Visweswaran, “A P1500 Compliant Programmable BistShell for Embedded Memories”, in Proc. IEEE International Workshop on Memory Technology, Design and Testing (MTDT), pp. 21-27, 2001

[30] C. W. Wang, C. F. Wu, J. F. Li, C. W. Wu, T. Teng, K. Chiu, and H. P. Lin, “A Built-In Self-Test and Self-Diagnosis Scheme for Embedded SRAM”, in Proc. of IEEE Asian Test Symposium (ATS), pp. 45-50, 2000

[31] D. Appello, F. Corno, M. Giovinetto, M. Rebaudengo, and M. S. Reorda, “A P1500 Compliant BIST-Based Approach to Embedded RAM Diagnosis”, in Proc. of IEEE Asian Test Symposium (ATS), pp. 97-102, 2001

[32] J. T. Chen, J. Rajski, J. Khare, O. Kebichi, and W. Maly, “Enabling Embedded Memory Diagnosis via Test Response Compression”, in Proc. of IEEE VLSI Test Symposium (VTS), pp. 292-298, 2001

[33] J. Dreibelbis, J. Barth, H. Kalter, and R. Kho, “Processor-Based Built-In Self-Test for Embedded DRAM”, Solid-State Circuits, Vol. 33, no.11, pp. 1731-1740, November 1998

[34] Y. Zorian and S. Shoukourian, “Embedded-Memory Test and Repair: Infrastructure IP for SoC Yield”, IEEE Design and Test of Computers, Vol. 20, no.3, pp. 58-66, May-June 2003

[35] J. C. Yeh, C. F. Wu, K. L. Cheng, Y. F. Chou, C. T. Huang, and C. W. Wu, “Flash Memory Built-In Self-Test Using March-Like Algorithms”, in Proc. of IEEE International Workshop on Electronic Design, Test and Applications, pp. 137-141, 2002

[36] Y. Zorian, “A Distributed BIST Control Scheme for Complex VLSI Devices”, in Proc. of IEEE VLSI Test Symposium (VTS), pp. 4-9, 1993

[37] M. L. Bodoni, A. Benso, S. Chiusano, S. D. Carlo, G. D. Natale, and P. Prinetto, “An Effective Distributed BIST Architecture for RAMS”, in Proc. of IEEE European Test Workshop (ETW), pp. 119-124, 2000

[38] R. Rajsuman, “Testing a System-on-a-Chip with Embedded Microprocessor”, in Proc. of IEEE International Test Conference (ITC), pp. 499-508, 1999

[39] C. H. Tsai, C. W. Wu, “Processor-Programmable Memory BIST for BUS- Connected Embedded Memories”, in Proc. of Asia and South Pacific, Design Automation Conference (ASP-DAC), pp. 325-330, 2001

[40] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis, "Software-Based Self-Testing of Embedded Processors," IEEE Transactions on Computers, Vol. 54, no. 4, pp. 461-475, 2005.

[41] A. Paschalis, D. Gizopoulos, “Effective software-based self-test strategies for on-line periodic testing of embedded processors”, IEEE Transactions on CAD, Vol. 24, no.1, pp.88 – 99, Jan. 2005.

[42] I. Bayraktaroglu, J. Hunt, and D. Watkins, ‘‘Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues,’’ in Proc. of IEEE International Test Conference (ITC), 2006.

[43] P. Parvathala, K. Maneparambil, W. Lindsay, “FRITS – A Microprocessor Functional BIST Method”, in Proc. of IEEE International Test Conference (ITC), 2002, pp.590-598.

[44] M. Psarakis, D. Gizopoulos, E. Sanchez, M. Sonza Reorda, "Microprocessor Software-Based Self-Testing", IEEE Design and Test of Computers, vol. 27, no. 3, pp. 4-19, May/June 2010.

[45] S.M. Thatte, J.A. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, Vol. 29, no. 6, pp. 429–441, 1980

[46] D. Brahme, J.A. Abraham, “Functional Testing of Microprocessors” IEEE Transactions on Computers, Vol. 33, no. 6, pp. 475–485, 1984

[47] J. Shen and J.A. Abraham, ‘‘Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation’’, in Proc. of IEEE International Test Conference (ITC), 1998, pp. 990-999

[48] K. Batcher, C. Papachristou, “Instruction Randomization Self-Test for Processor Cores”, in Proc. of VLSI Test Symposium (VTS), 1999, pp 34–40

[49] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero, “Automatic Test Program Generation: A Case Study”, IEEE Design & Test, vol. 21, no. 2, pp. 102-109, 2004

[50] L. Chen, S. Dey, “Software-Based Self-Testing Methodology for Processor Cores”, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol 20m no 3, pp. 369–380, 2001

[51] L. Chen, S. Ravi, A. Raghunathan, S. Dey, “A Scalable Software-Based Self-Testing Methodology for Programmable Processors”, In Proc. of Design Automation Conference, 2003, pp. 548–553

[52] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, “ On the Test of Microprocessor IP Cores” In Proc. of Design Automation & Test in Europe (DATE), 2001, pp 209–213

[53] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, “Fully Automatic Test Program Generation for Microprocessor Cores”, In Proc. of Design Automation & Test in Europe, 2003, pp 1006-1011

[54] K. Kambe, M. Inoue, H. Fujiwara, “Efficient Template Generation for Instruction-Based Self-Test of Processor Cores”, In Proc. of IEEE Asian Test Symposium (ATS), 2004, pp. 152–157

[55] N. Kranitis, A. Paschalis, D. Gizopoulos, Y. Zorian, “Instruction-Based Self-Testing of Processor Cores”, Journal of Electronic Testing: Theory and Applications, no 19, pp 103–112, 2003

[56] E. Sanchez, M.S. Reorda, G. Squillero, “On the Transformation of Manufacturing Test Sets into On-Line Test Sets for Microprocessors”, In Proc. of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), 2005, pp 494–502

[57] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, S. Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, In Proc. of Design Automation Conference (DAC), 2006, pp 393–398

[58] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, S. Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 16, no. 11, pp. 1441-1453, 2008

[59] N. Kranitis, A. Merentitis, N. Laoutaris, G. Theodorou, A. Paschalis, D. Gizopoulos, C. Halatsis, “Optimal Periodic Testing of Intermittent Faults In Embedded Pipelined Processor Applications”, In Proc. Design Automation & Test in Europe (DATE), 2006, pp 65–70

[60] WC. Lai, A. Krstic, KT. Cheng, “Functionally Testable Path Delay Faults on a Microprocessor”, IEEE Design and Test of Computers, Vol. 17, no. 6, pp. 6–14, 2000

[61] A. Krstic, L. Chen, WC. Lai, KT. Cheng, S. Dey, “Embedded Software-Based Self-Test for Programmable Core-Based Designs”, IEEE Design and Test of Computers, Vol. 19, no. 4, pp. 18–26, 2002

[62] V. Singh, M. Inoue, KK. Saluja, H. Fujiwara, “Instruction-Based Self-Testing of Delay Faults in Pipelined Processors”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, no. 11, pp.1203–1215, 2006

[63] S. Gurumurthy, S. Vasudevan, J.A. Abraham, “Automated Mapping of Pre-Computed Module-Level Test Sequences to Processor Instructions”, In Proc. of IEEE International Test Conference (ITC), 2005, pp 294–303

[64] S. Gurumurthy, S. Vasudevan, J.A. Abraham, “Automatic Generation of Instruction Sequences Targeting Hard-to-Detect Structural Faults in a Processor”, In Proc. of IEEE International Test Conference (ITC), 2006, paper 27.3

[65] C.H.P. Wen, L.C. Wang, K.T Cheng, W.T. Liu, J.J. Chen, “Simulation-Based Target Test Generation Techniques for Improving the Robustness of a Software-Based-Self-Test Methodology”, In Proc. of IEEE International Test Conference (ITC), 2005, pp 936-945

[66] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, D. Gizopoulos, "Hybrid-SBST Methodology for Efficient Testing of Processor Cores", IEEE Design & Test of Computers, Vol. 25, no. 1, pp.64-75, 2008

[67] A. Apostolakis, D. Gizopoulos, M. Psarakis, A. Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors,", IEEE Transactions on Computers, Vol.58, no.12, pp.1682-1694, 2009

[68] N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, A. Gonzalez, "MT-SBST: Self-test optimization in multithreaded multicore architectures," in Proc. of IEEE International Test Conference (ITC), 2010, pp.1-10

[69] A.J. van de Goor, T.J.W Verhallen, "Functional testing of current microprocessors", in Proc. of IEEE International Test Conference (ITC), 1992, pp.684-695.

[70] J. Sosnowski, "In-system testing of cache memories", in Proc. of IEEE International Test Conference (ITC), 1995, pp.384-393.

[71] S.M. Al-Harbi, S.K. Gupta, "A Methodology for Transforming Memory Tests for In-System Testing of Direct Mapped Cache Tags", in Proc. of VLSI Test Symposium (VTS), 1998, pp.394-400.

[72] J. Sosnowski, "Improving Software Based Self-Testing for Cache Memories," in Proc. of Design and Test Workshop (DTW), 2007.

[73] M. Tuna, O. Garcia, M. Benabdenbi, "Software-Based Self-Test Strategies for Memory Caches of RISC Processor Cores", in Proc. of IEEE Latin-American Test Workshop, 2007, pp.124-130.

[74] S. Alpe, S. Di Carlo, P. Prinetto, A. Savino, "Applying March Tests to K-Way Set-Associative Cache Memories", in Proc. of the 13th European Test Symposium (ETS), 2008, pp.77-83.

[75] Y.C. Lin, Y.Y. Tsai, K.J. Lee, C.W. Yen, C.H. Chen, "A Software-Based Test Methodology for Direct Mapped Data Cache", in Proc. of Asian test Symposium, 2008, pp.363-368.

[76] W.J. Perez, J. Velasco, D. Ravotto, E. Sanchez, M. Sonza Reorda, "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs", in Proc. of IEEE International On-Line Testing Symposium (IOLTS), 2008, pp.143-148.

[77] W.J. Perez, D. Ravotto, E. Sanchez, M. Sonza Reorda, A. Tonda, "On the Generation of Functional Test Programs for the Cache Replacement Logic", in Proc. of IEEE Asian Test Symposium (ATS), 2009, pp.418-423.

[78] A.J. van de Goor, G. Gaydadjiev, S. Hamdioui, "Memory testing with a RISC microcontroller", in Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pp.214-219

[79] S. Di Carlo, P. Prinetto, A. Savino, "Software-Based Self-Test of Set-Associative Cache Memories," IEEE Transactions on Computers, Vol.60, no.7, pp.1030-1044, July 2011

[80] B. Jacob, S.W. Ng, D. T. Wang, "Memory Systems: Cache, DRAM, Disk", Burlington. USA : Morgan Kaufmann, 2008

[81] MIPS Architecture for programmers. Available online: www.mips.com

[82] ARM Architecture reference manual. Available online: www.arm.com

[83] SPARC Assembly reference manual. Available online: www.oracle.com

[84] Intel® 64 and IA-32 Architectures Software Developer's Manual, Available online: www.intel.com

[85] A.J. van de Goor, I.B.S. Tlili, "March Tests for Word-Oriented Memories" in Proc. of the Design, Automation and Test in Europe (DATE), 1998, pp.501-508

[86] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, Y. Zorian, "Generic BIST architecture for testing of content addressable memories," in Proc. of IEEE International On-Line Testing Symposium (IOLTS), 2011, pp.86-91

[87] C. Kim, D. Burger, S.W. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches", IEEE Micro, vol.23, no.6, pp. 99- 107, Nov.-Dec. 2003

[88] C.F. Wu et al, "RAMSES: a fast memory fault simulator", in Proc. of the International Symposium on Defect and Fault Tolerance in VLSI Systems, 1999, pp.165-173

[89] M. Lin, Ch. Yunji, S. Menghao, Q. Zichu, Zh. Heng, H. Weiwu, "Testing Content Addressable Memories Using Instructions and March-Like Algorithms", in Proc. of IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2008

[90] P. Manikandan, B. B. Larsen, E. J. Aas, S. M. Reddy, "Test of Embedded Content Addressable Memories", in Proc. of IEEE Int'l Symposium on Electronic System Design (ISED), 2010

[91] Said Hamdioui, "Testing Static Random Access Memories: Defects, Fault Models and Test Patterns", Kluwer Academic Publishers, USA, Boston, 2004

[92] Jari Nurmi, "Processor Design: System-On-Chip Computing for ASICs and FPGAs", Springer, The Netherlands, 2010

[93] Laung-Terng Wang, Cheng-Wen Wu, Xiaoqing Wen, "VLSI Test Principles and Architectures: Design for Testability", Elsevier, Morgan Kaufmann Publishers, San Franscisco, USA, 2004

[94] Laung-Terng Wang, Charles E. Stroud, Nur A. touba "System-on-Chip Test Architectures: Nanometer Design for Testability", Elsevier, Morgan Kaufmann Publishers, Burlington, USA, 2008

[95] David A. Patterson, John L. Hennessy, "Computer Organization and Design, Fourth Edition: The Hardware/Software Interface", Elsevier, Morgan Kaufmann Publishers, Burlington, USA, 2009