Institut für Technische Informatik
Universität Stuttgart
Pfaffenwaldring 47
D-70569 Stuttgart

Bachelorarbeit Nr. 183

Software-basierter Selbsttest eingebetteter Speicher

Felix Ebinger

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer/in:	Dipl.-Inf. Dominik Ull
Beginn am:	21. Oktober 2015
Beendet am:	21. April 2015
CR-Nummer:	B.3.4

Kurzfassung

Prozessoren werden häufig mittels softwarebasierter Selbsttests (SBST) getestet, da dieses Testverfahren mehrere Vorteile besitzt. Zunächst ist der Test zerstörungsfrei, und wird im funktionalen Betriebszustand des Prozessors durchgeführt. Es ist weder eine Veränderung des Hardwaredesigns erforderlich noch ist ein Übertesten möglich. Die Testmethode ist flexibel einsetzbar und kann sowohl beim Herstellungstest als auch im Feld genutzt werden. Speicher werden dagegen üblicherweise mittels eingebauter Selbsttests (engl. built-in self-test, BIST) getestet, da der Overhead durch die zusätzliche Testhardware nur gering ausfällt und diese Tests bei Speichern ohne Performance-Einbußen realisiert werden können.

In dieser Arbeit wird die softwarebasierte Umsetzung von Speichertests untersucht um die Vorteile softwarebasierter Selbsttests auch bei Speichertests nutzen zu können. Dies stellt eine Herausforderung dar, da softwarebasiert nicht jede Operationsfolge mit frei wählbarem Zeitverhalten erzeugt werden kann. Insbesondere bei dynamischen Fehlern kann dies zu einer Verringerung der Testabdeckung führen.

Hierzu wird ein Framework zur automatischen Umwandlung von Marchtestbeschreibungen in Testprogramme für den miniMIPS-Prozessor vorgestellt. Dabei steht besonders die Laufzeit des Testprogramms und die erreichte Testabdeckung im Vordergrund. Die Testabdeckung wird durch Simulation und Fehlerinjektion experimentell bestimmt. Es zeigt sich, dass die Fehlerabdeckung für die untersuchten statische und dynamische Fehlermodelle durch die vorgestellte Implementierung in Software nicht beeinträchtigt wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation

Softwarebasierte Selbsttests sind eine erprobte und bewährte Testmethode für Hardware. Seit über drei Jahrzehnten wird sie genutzt um beispielsweise Mikroprozessoren[PGSR10][PGH+06], Caches[TKPG10][TKPG11] oder Peripheriekomponenten[GHS+12][AGP+09] zu testen. Ihre Verbreitung verdankt sie vor allem den Vorteilen, die sie gegenüber anderen Hardwaretestverfahren auszeichnet. Dazu gehört insbesondere, dass das Hardwaredesign nicht für die Tests angepasst werden muss. Die Tests erfolgen konstruktionsbedingt bei Betriebsfrequenz und können, da sie in Software implementiert sind, im Nachhinein verändert und erweitert werden. Die Testmethode ist außerdem flexibel einsetzbar, sie kann nicht bei Fertigungstests, sondern auch im Feld eingesetzt werden.

Speicher werden üblicherweise mittels in die Hardware integrierter Testschaltungen getestet. Durch die einfache und reguläre Struktur von Speicherchips, entsteht durch die zusätzliche Testschaltungen nur ein geringer Hardware-Overhead[BA00]. Mittels integrierter Testschaltungen lässt sich die Testdauer im Vergleich zu konventionellen Hardwaretestverfahren um eine Größenordnung reduzieren[Goo91].

Ziel dieser Arbeit ist die Untersuchung, ob existierende March-Speichertests softwarebasiert durchführbar sind. Wichtige Aspekte dieser Untersuchung sind die Laufzeit des Tests und die Fehlerabdeckung. Softwarebasierte Selbsttests erlauben ohne zusätzlich integrierte Testinfrastruktur eine flexible Anwendung von Speichertests.

Dazu wird im Rahmen dieser Arbeit zunächst ein Framework entwickelt, das für RISC-Architekturen existierende Marchtestbeschreibungen in softwarebasierte Testprogramme umwandelt. Für dynamische Speicherfehler, deren Aktivierung von bestimmten Operationsfolgen oder Zeitverhalten abhängig ist, muss die Fehlerabdeckung experimentell bestimmt werden und überprüft werden, inwieweit die Fehlerabdeckung der Tests durch die Implementierung in Software beeinträchtigt wird. Für statische Fehler ist diese experimentelle Überprüfung nicht notwendig, da die Aktivierung dieser Fehler nicht vom Zeitverhalten oder von einer bestimmten Befehlsfolge abhängig ist.

Bei softwarebasierten Selbsttests wird der Mikroprozessor als Testgerät genutzt. Dadurch entstehen, verglichen mit speziell für die Tests konstruierter Hardware, Einschränkungen, da für die Tests nur der Befehlssatz des Prozessors genutzt werden kann. Mit diesem kann nur über die Schnittstelle auf die zu testende Komponente zugegriffen werden, während integrierte Testschaltungen dieser Einschränkung nicht unterliegen.

Diese Einschränkungen nennt man funktionale Nebenbedingungen. Dadurch ist es nicht immer möglich jede Testoperation direkt in Software umzusetzen, stattdessen müssen diese durch Abfolge von Softwarebefehlen emuliert werden. Außerdem besteht Software im Allgemeinen noch aus

Kontrollstrukturen, das heißt aus zusätzlichen Instruktionen, die ausgeführt werden müssen. Beides verlängert die Testausführung, weshalb die Laufzeit der in Software implementierten Tests untersucht werden muss.

Durch die bereits erwähnten funktionalen Nebenbedingungen entsteht außerdem die Problematik, dass manche Testanforderungen, die eine direkte Hintereinanderausführung bestimmter Testbefehle erfordern, nicht mehr eingehalten werden können. Dies wirkt sich direkt auf die Fehlerabdeckung und damit die Qualität des Testes aus. Eine Herausforderung ist es daher, durch geeignete Assemblertemplates die Auswirkungen dieser Nebenbedingungen zu minimieren. Die mittels dieser Templates umgesetzten Tests müssen dann bezüglich ihrer Fehlerabdeckung untersucht werden.

1.2 Überblick

In Kapitel 2 werden zunächst die Grundlagen erklärt. Es wird zuerst die Unterscheidung zwischen funktionalen und strukturellen Tests (Kapitel 2.1) und die Testmethode softwarebasierter Selbsttests (Kapitel 2.2) erklärt. Danach werden verschiedene Speichermodelle (Kapitel 2.3) und die in dieser Arbeit genutzten Fehlermodelle (Kapitel 2.4), sowie die Speichertestfamilie Marchtests (Kapitel 2.5) vorgestellt.

In Kapitel 3 wird die Implementierung der Arbeit erläutert. Dabei wird zuerst das entwickelte Framework zur Testgenerierung (Kapitel 3.1) vorgestellt, danach wird die experimentelle Bestimmung der Fehlerabdeckung durch Fehlerinjektion erklärt (Kapitel 3.2).

In Kapitel 4 werden die Ergebnisse der Arbeit vorgestellt. In Kapitel 4.1 werden zunächst die Untersuchungsergebnisse bezüglich der Effizienz erläutert und in Kapitel 4.2 dann die Ergebnisse bezüglich der Fehlerabdeckung. Die Arbeit schließt mit einem Fazit in Kapitel 5 und einem Ausblick in Kapitel 6.

2 Grundlagen

2.1 Funktionales und strukturelles Testen

Beim Testen von Hardwarekomponenten unterscheidet man prinzipiell zwischen funktionalen und strukturellen Ansätzen. Sie unterscheiden sich in der Art an Informationen, die zur Testgenerierung verwendet werden[BA00].

Funktionales Testen Bei funktionalen Tests wird die Funktionalität einer Komponente unabhängig von ihrer internen Struktur getestet. Es wird nur das Verhalten nach außen hin überprüft, daher spricht man auch von einem Black-Box-Test. Für diese Tests betrachtet man nur die Ausgänge der Komponente und legt an den Eingängen Testmuster, sogenannte Testvektoren, an[Wun09]. Um einen vollständigen funktionalen Test durchzuführen, also sicherzustellen, dass eine Komponente sich in jedem Anwendungsfall korrekt verhält, muss jedes mögliche Eingabemuster als Testmuster anlegt und das Ergebnis mit dem entsprechenden Sollwert verglichen werden. Betrachtet man nun z.B. eine arithmetisch-logische Einheit einer CPU mit zwei Dateneingängen mit jeweils 32 Bit Wortbreite, so sind bereits 2^{64} Testmuster notwendig. Es wird deutlich, dass vollständiges funktionales Testen in annehmbarer Testzeit praktisch nicht durchführbar ist. Bei sequentiellen Schaltungen hängt die Testbarkeit eines Fehlers zusätzlich vom internen Zustand der Schaltung ab.

Strukturelles Testen Bei strukturellen Tests werden zusätzlich zu funktionalen Informationen auch Informationen über den internen Aufbau der Komponente genutzt, man spricht bei diesen Tests daher auch von White-Box-Tests. Dazu werden Hardwaredefekte mithilfe von Fehlermodellen beschrieben. Diese Fehlermodelle erlauben die Angabe einer Gesamtzahl von Fehlern und ermöglichen so die Bestimmung einer Fehlerabdeckung und damit ein Maß für die Testqualität. Da das Augenmerk dieser Tests jedoch nicht mehr nur auf dem Testen der Funktionalität liegt, sondern auf dem Erreichen einer möglichst hohen Fehlerabdeckung, ist es möglich, dass Testmuster entstehen, die im realen Einsatz der Komponente nicht auftreten können. Dadurch können Komponenten, die funktional gesehen korrekt funktionieren als defekt klassifiziert werden, wodurch die Ausbeute verringert wird (engl. yield lost). Dies wird als Übertesten (engl. overtesting) bezeichnet.

Je nach Anwendungsfall ist die Unterscheidung zwischen rein funktionalen und rein strukturellen Tests nicht immer möglich.

2.2 Software-basierter Selbsttest

Die klassischen Ansätze für den Hardwaretest benötigen entweder sehr teure Testgeräte (engl. automatic test equipment, ATE) um die Testmuster an die zu testende Komponente (engl. circuit under test, CUT) anzulegen oder zusätzliche, integrierte Testschaltungen. Die ATEs sind nicht nur sehr teuer, sondern veralten auch sehr schnell, wenn man die Entwicklungsgeschwindigkeit in der Hardwareentwicklung betrachtet. Typischerweise werden diese Testmuster auch nicht mit dem Systemtakt angelegt, sondern langsamer, wodurch zeitkritische Fehler nicht entdeckt werden können. Integriert man dagegen direkt in die Komponente zusätzliche Testschaltungen, so erfordert dies ein für den Test angepasstes Hardwaredesign, benötigt zusätzlichen Platz auf dem Chip und beeinflusst das Zeitverhalten der Komponente.

Während bei BIST das System in einem speziellen, nicht funktionalem Testmodus betrieben wird, wird beim softwarebasierten Selbsttest das System im normalen Betriebsmodus genutzt, diese Tests sind also zerstörungsfrei (engl. non-intrusive testing). Der Test wird dabei als Anwendung betrachtet und es wird die Tatsache ausgenutzt, dass Mikroprozessoren programmierbar sind[PGSR10]. Da diese Tests im regulären Betriebsmodus erfolgen, werden sie folglich auch bei regulärem Systemtakt ausgeführt (engl. at-speed testing). Dadurch, dass vorhandene Systemkomponenten zum Test genutzt werden, wird keine zusätzliche Testhardware benötigt, das Design muss also nicht für den Test angepasst werden. Des Weiteren werden dadurch die Anforderungen an die externen Testgeräte reduziert, da diese höchstens dafür benötigt werden, den Test in den Speicher zu laden und später die Testergebnisse im Speicher zu überprüfen. Ein weiterer Vorteil ist, dass die Tests auch später noch geändert werden können und nicht bereits durch entsprechende Hardwarestrukturen festgelegt oder zumindest eingeschränkt werden.

Die Testmethode der softwarebasierten Selbsttests erlaubt eine sehr flexible Anwendung, sie kann nicht nur als Herstellungstests genutzt, sondern auch im Feld angewandt werden. Diese Tests können auch während das System läuft ausgeführt werden und können so z.B. während das System nicht genutzt wird, die Funktionalität der Hardware zu überprüfen. Im Gegensatz zu anderen strukturellen Testmethoden besteht bei softwarebasierten Selbsttests aufgrund der funktionalen Testdurchführung auch nicht die Gefahr des Übertestens.

Der prinzipielle Ablauf von softwarebasierten Selbsttests ist unabhängig von der getesteten Hardwarekomponente stets gleich:

1. Generierung des Testprogramms
2. Testprogramm in Speicher laden
3. Testprogramm ausführen
4. Testergebnisse auslesen und vergleichen

Auch bei Software basierten Selbsttests unterscheidet man zwischen dem funktionalen und strukturellen Ansatz[PGSR10]. Während beim funktionalen Ansatz nur funktionale Informationen bei der Testerzeugung genutzt werden (z.B. Befehlssatz), werden hierzu beim strukturellen Ansatz auch Strukturinformationen (z.B. Beschreibungen auf Gatterebene oder Registertransferebene) genutzt. Diese Informationen ermöglichen die Bestimmung einer strukturellen Testabdeckung. Erst dadurch

wird es möglich, qualitative Aussagen über Tests zu machen und Tests bezüglich ihrer Effizienz zu bewerten. Die Vorteile des funktionalen Ansatzes dagegen ergeben sich aus der Einschränkung an nutzbaren Informationen zur Testgenerierung. So ist es auch möglich, Tests zu generieren, wenn Strukturinformationen nicht verfügbar sind.

2.2.1 Speichertests mittels Software-basierten Selbsttests

Speicher werden typischerweise mittels BIST getestet. Auf diese Weise kann die Testzeit um eine Größenordnung reduziert werden[Goo91]. Durch die reguläre Struktur von Speicherkomponenten entsteht hierbei nur ein geringer Hardware-Overhead von 2%[BA00]. Darüber hinaus ist die Integration von Teststrukturen bei Speichern ohne kritische Performance-Einbußen möglich.

Im Vergleich zu BIST kann SBST flexibler angewendet werden[RGT+14]. Die Durchführung von BIST setzt einen dedizierten Testmodus und einen internen oder externen Testcontroller voraus. Besonders in sicherheitskritischen Anwendungen existieren strenge Rahmenbedingungen für die Aktivierung des Testmodus im Feld. Bei SBST ist dies nicht der Fall, da diese Tests im funktionalen Betriebsmodus ausgeführt werden, wobei ein existierender Prozessor im System als Testcontroller dient. Da der Betriebsmodus nicht gewechselt werden muss„ kann SBST auch abschnittsweise oder begrenzt auf einen bestimmten Speicherbereich im Betrieb durchgeführt werden. Da bei softwarebasierten Selbsttests nur die Speicherschnittstelle des Prozessors und der Befehlssatz des Prozessors genutzt werden, besteht zwischen dem Testen eingebetteter Speicher und Speichern, die auf einem eigenen Chip untergebracht sind, kein Unterschied.

Es existieren bisher nur wenige Untersuchungen, die sich mit der softwarebasierten Ausführung von Speichertests befassen. Eine Arbeit über Tests von System-on-a-Chips mittels softwarebasierter Selbsttests[Raj99] führt auch einen einfachen softwarebasierten Speichertest ein. Dieser beschränkt sich auf die Untersuchung von Haftfehlern und die Erzeugung des Testprogramms erfolgt manuell. Eine weitere Arbeit beschäftigt sich mit dem Testen auf statische Fehler mittels SBST[GGH10], beschränkt die Untersuchung aber insbesondere auf die spezielle Speicherarchitektur des zu testenden Mikrocontrollers und die manuelle Erzeugung des Testprogramms.

Mehr Untersuchungen beschäftigen sich damit, inwieweit Caches mittels softwarebasierter Selbsttests testbar sind[TKPG10][TKPG11]. Obwohl Caches auch Speicher sind, unterscheidet sich ihr Test vom Test normaler Speicher. Caches sind sehr klein, sodass sich der Overhead durch die zusätzliche Testhardware signifikant auswirkt, insbesondere da Caches normalerweise auf dem selben Chip wie der Mikroprozessor selbst untergebracht sind[TKPG10]. Da an die Antwortzeit von Caches besondere Anforderungen gestellt werden, wirkt sich auch der Einfluss der Testschaltungen negativ auf das Timing aus. Daher wird für Cache-Tests auf SBST zurückgegriffen. Beim Testen von Caches bestehen andere Herausforderungen als beim Testen von Speichern. Caches bestehen aus zwei Speichern, zum einen dem Datenspeicher und zum anderen dem Speicher für Adress-Tags. Beide müssen getestet werden, auf beide hat besteht jedoch durch einen typischen Befehlssatz kein direkter Zugriff[TKPG11]. Außerdem muss die Organisation der Caches beachtet werden, damit der ganze Cache getestet werden kann. Daher sind die für Caches erprobte Testmethoden im Allgemeinen nicht auf Speicher übertragbar.

2.3 Speichermodelle

Speichermodelle werden genutzt um die komplexen Speicherchips vereinfacht und strukturiert darzustellen. Durch ein Speichermodell führt man eine Abstraktionsebene ein um die Implementierungsdetails, die für den jeweiligen Anwendungszweck nicht benötigt werden, zu verstecken. Das Modell wird dabei auf die für den jeweiligen Einsatzzweck relevanten Attribute und Eigenschaften beschränkt. Die hier vorgestellten Speichermodelle dienen alle dem Testen von Speichern. Es existieren verschiedene Ansätze Speicher zu modellieren, die sich durch unterschiedlich genaue Abbildung der realen Komponente, also im Abstraktionslevel, unterscheiden[Goo91]:

Zunächst gibt es das Verhaltensmodell (engl. behavorial model), es basiert ausschließlich auf der Spezifikation des Speichers und nutzt keine Informationen über den tatsächlichen Systemaufbau, man spricht daher auch von einem Black-Box-Modell. Dieses Modell ist technologieunabhängig und ist das abstrakteste vorgestellte Modell. Ein typischer Test auf Basis dieses Modells ist ein Kompletttest, bei dem jedes mögliche Datenwort an jede Adresse geschrieben und gelesen wird, was bei der Größe heutiger Speicher jedoch unpraktikabel ist.

Weniger abstrakt ist das funktionale Modell (engl. functional model). Auch dieses Modell basiert auf der funktionalen Spezifikation, es werden jedoch zusätzlich Annahmen über den internen Aufbau des Speichers getroffen. Die interne Struktur ist also teilweise sichtbar, daher spricht man auch von einem Gray-Box-Modell. Das auf diesem Modell basierende funktionale Testen validiert die Funktionalität des Systems auf Basis der getroffenen Annahmen, wie der Existenz bestimmter funktionaler Blöcke (z.B. Speicherzellen), über den internen Aufbau. Beschränkt man das Einsatzgebiet auf die Fehlererkennung und verzichtet auf die Lokalisierung des Fehlers, so kann auf ein funktionales Speichermodell mit nur wenigen Annahmen über den internen Aufbau zurückgegriffen werden. Dieses Modell wird auch als vereinfachtes funktionales Speichermodell bezeichnet (engl. reduced functional model) und besteht aus drei Komponenten, dem Adressdekoder, der Lese-/Schreiblogik und der Speicherzellen.

Ein weiterer typischer Modellierungsansatz für Hardware ist das logische Modell. Da ein Speicher jedoch - abgesehen von der Adressierung - nicht aus logischen Gattern besteht, ist dieses Modell für Speichertests nicht relevant.

Setzt man voraus, dass die interne Struktur des Speichers komplett bekannt ist, so spricht man von White-Box-Modellen. Zum einen gibt es dabei das elektronische Modell (engl. electrical model), dieses Modell basiert auf dem Wissen über den internen Aufbau auf elektronischer Ebene, sowie auf der funktionalen Spezifikation und beschreibt, wie diese mittels elektronischer Bauelemente umgesetzt wird. Im Gegensatz zu den bisherigen Modellen kann dabei die fehlerhafte Komponente genau lokalisiert werden, da nun der gesamte interne Aufbau bekannt ist. Dieses Wissen erlaubt es außerdem die einzelnen Komponenten sehr effizient zu testen, da bekannt ist, welche Fehler wo auftreten können.

Nimmt man zusätzlich das Wissen über das Layout des Chips hinzu, also das Wissen wo welche Komponente auf dem Chip platziert ist und wie diese verdrahtet ist, so spricht man von einem geometrischen Modell (engl. geometrical model). Dieser Detailgrad wird benötigt um Fehler im Herstellungsprozess zu erkennen oder auch um zum Beispiel Alterungsprozesse auf einem Chip modellieren zu können.

2.4 Vereinfachte funktionale Fehlermodelle

Um physikalische Defekte einer Schaltung durch die Beobachtung des logischen Schaltverhaltens erkennen zu können, müssen diese Fehler auf logische Fehlermodelle abgebildet werden. Im Folgenden werden ausgewählte vereinfachte funktionale Fehlermodelle (engl. reduced functional fault models) vorgestellt, diese basieren auf dem vereinfachten funktionalen Speichermodell.

Die vorgestellten Fehlermodelle lassen sich prinzipiell in zwei Gruppen unterteilen. Zum einen gibt es Fehler, bei denen nur eine Speicherzelle betroffen ist (z.B. Haftfehler, Transitionsfehler) und zum anderen gibt es Fehler bei denen zwei Speicherzellen betroffen sind (z.B. Kopplungsfehler).

Eine weitere Kategorisierung der Fehler ist die Unterscheidung zwischen statischen und dynamischen Fehlern. Statische Fehler sind immer aktiv, während dynamische Fehler nur aktiviert werden, wenn z.B. eine bestimmte Operationsfolge ausgeführt wird oder der Zugriff auf gespeicherte Daten erst nach einer bestimmten Zeit erfolgt.

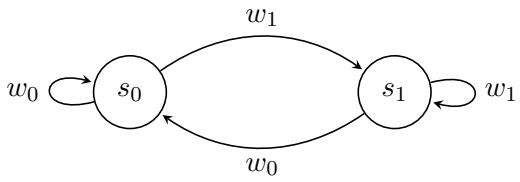
Zunächst werden die verschiedenen Fehlermodelle für die Speicherzellen erklärt. Die selben statischen Fehler können dabei auch in der Lese/Schreiblogik auftreten, allerdings werden sie durch die selben Tests, die diese Fehler in den Speicherzellen erkennen, erkannt und müssen daher nicht extra getestet werden[Goo91]. Allerdings ermöglichen die Tests es nicht, zwischen Fehlern in der Lese-/Schreiblogik und den Fehlern im Speicher selbst zu unterscheiden.

2.4.1 Statische Fehler

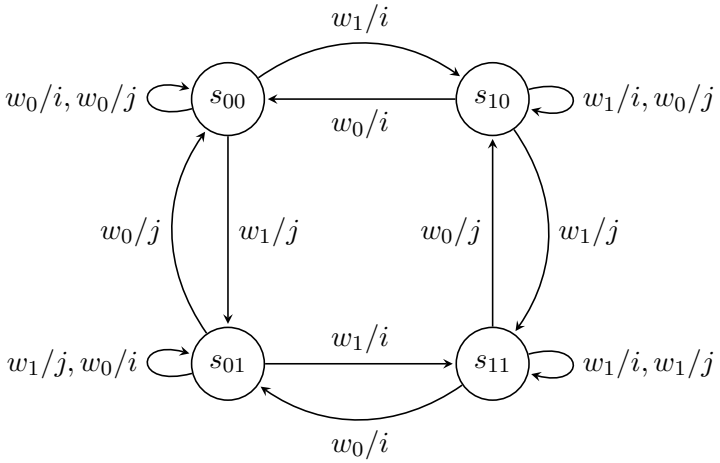
Das Verhalten einer Speicherzelle lässt sich durch einen Zustandsautomaten beschreiben[Goo91]. Eine Zelle wird dabei durch zwei Zustände modelliert, in s_0 ist in der betreffenden Zelle eine `0` gespeichert, in Zustand s_1 eine `1`. Die Zustandsübergänge erfolgen durch Schreiboperationen. Abbildung 2.1a zeigt das Modell einer funktionierenden Speicherzelle, Abbildung 2.1b zeigt das Modell zweier funktionierender Speicherzellen. Der Zustand s_{ij} stellt dabei immer den Inhalt der Speicherzellen i und j dar, der bei den Übergängen angegebene Buchstabe gibt an, in welche Zelle geschrieben wird. Um mit diesem Modell n Zellen zu modellieren werden 2^n Zustände benötigt. Die Modellierung wird daher auf möglichst wenige Zellen beschränkt und es wird die reguläre Struktur von Speichern ausgenutzt um dieses Modell zu erweitern[Wun09].

Haftfehler

Allgemein bezeichnet ein Haftfehler (engl. stuck-at fault, SAF), dass bestimmte Gattereingänge oder -ausgänge ständig den logischen Wert 1 oder 0 haben. Bei Haftfehlern in Speichern unterscheidet man zwei verschiedene Haftfehler, zum einen die Stuck-At-Zero Fehler, bei denen in der Speicherzelle immer eine `0` gespeichert ist, unabhängig vom tatsächlich gespeicherten Wert und zum anderen die Stuck-At-One Fehler, bei denen immer eine `1` in der Zelle gespeichert ist. Abbildung 2.2a zeigt das Zustandsdiagramme einer Speicherzelle mit einem Stuck-At-Zero Fehler, während Abbildung 2.2b das Zustandsdiagramm mit einem Stuck-At-One Fehler zeigt. In beiden Fällen ist der Übergang in den jeweiligen anderen Zustand nicht möglich.



(a) Zustandsautomat für eine Speicherzelle



(b) Zustandsautomat für zwei Speicherzellen

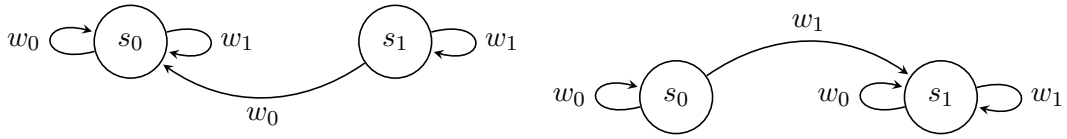
Abbildung 2.1: Zustandsautomat funktionierende Speicherzelle

(a) Stuck-At-Zero Fehler **(b)** Stuck-At-One Fehler

Abbildung 2.2: Speicherzelle mit Haftfehler

Transitionsfehler

Wenn eine Speicherzelle nicht in der Lage ist, den Übergang von einem gespeicherten Wert zum anderen zu vollziehen, so spricht man von einem Transitionsfehler (engl. transition fault, TF). Ist eine Speicherzelle nicht in der Lage einen Übergang von einer gespeicherten `0` zu einer `1` zu vollziehen (Transition `0`→`1`), so spricht man von einem steigenden Transitionsfehler (engl. up transition fault), ist der Übergang von einer gespeicherten `1` zu einer `0` nicht möglich (Transition `1`→`0`), so spricht man von einem fallenden Transitionsfehler (engl. down transition fault). Abbildung 2.3a zeigt das Zustandsdiagramm einer Zelle mit steigendem Transitionsfehler, Abbildung 2.3b das mit fallendem Transitionsfehler. Sobald die Zelle beim steigenden Transitionsfehler den Zustand s_0 erreicht hat, ist sie in diesem gefangen und kann ihn nicht mehr verlassen, analog beim fallenden Transitionsfehler in Zustand s_1.



(a) Steigender Transitionsfehler **(b)** Fallender Transitionsfehler

Abbildung 2.3: Speicherzelle mit Transitionsfehler

Kopplungsfehler

Bei Kopplungsfehlern (engl. coupling faults, CF) ist das Verhalten einer Speicherzelle an eine andere Speicherzelle gekoppelt. Die abhängige Zelle wird dabei als gekoppelte Zelle, im Englischen auch als Opferzelle (victim cell), bezeichnet. Die Zelle, die die gekoppelte Zelle dominiert, wird als koppelnde Zelle oder als Aggressorzelle bezeichnet. Man unterscheidet zwischen verschiedene Arten von Kopplungsfehlern[Goo91]:

Inversion coupling fault (CFin) Bei einem Inversions-Kopplungsfehler löst eine Transition in der Aggressorzelle eine Inversion des gespeicherten Wertes in der gekoppelten Zelle aus. Man unterscheidet zwischen zwei Inversions-Kopplungsfehlern, je nachdem, ob die `0`→`1` Transition den Fehler auslöst oder die `1`→`0` Transition. Abbildung 2.4 zeigt beispielhaft einen Inversions-Kopplungsfehler, der durch die Transition `0`→`1` in der Zelle i ausgelöst wird und dabei den gespeicherten Wert in der Zelle j invertiert.

Idempotent coupling fault (CFid) Bei einem idempotenten Kopplungsfehler erzwingt eine Transition in der Aggressorzelle einen bestimmten Wert (`0` oder `1`) in der gekoppelten Zelle. Man unterscheidet zwischen vier Typen von idempotenten Kopplungsfehler, je nachdem, ob die `0`→`1` Transition den Fehler auslöst oder die `1`→`0` Transition und jeweils, ob die gekoppelte Zelle den Wert `0` oder `1` annimmt. Abbildung 2.5 zeigt beispielhaft einen idempotenten Kopplungsfehler, der durch die Transition `0`→`1` in der Zelle i ausgelöst wird und dabei in der Zelle j den Wert `0` erzwingt.

State coupling fault (CFst) Beim Zustand-Kopplungsfehler ist nicht die Transition der Aggressorzelle, sondern ihr Zustand entscheidend. Dabei wird der gekoppelten Zelle ein bestimmter Wert aufgezwungen. Daraus ergeben sich wiederum vier unterschiedliche Zustand-Kopplungsfehler, je nachdem, ob der Zustand der Aggressorzelle `0` oder `1` ist und jeweils, ob die gekoppelte Zelle den Wert `0` oder `1` annimmt.

Disturb coupling fault (CFdst) Beim Störungs-Kopplungsfehler ist weder die Transition noch der Zustand der Aggressorzelle, sondern die Operation, die auf ihr ausgeführt wird, entscheidend. Dabei wird durch eine Operation auf der Agressorzelle die gekoppelte Zelle zu einer Transition gezwungen[GGMY96]. Bei vier möglichen Operationen und zwei möglichen Transitionen ergeben sich acht unterschiedliche störende Kopplungsfehler, je nachdem, welche der vier Operationen auf der Aggressorzelle ausgeführt wird und welche Transition dadurch in der gekoppelten Zelle erzwungen wird.

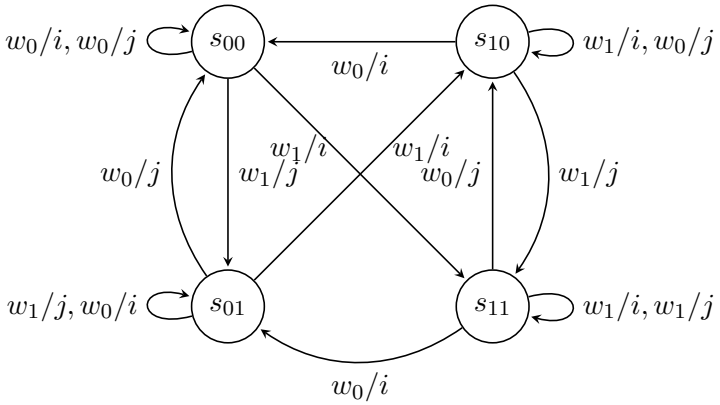


Abbildung 2.4: Speicherzellen mit einem Inversions-Kopplungsfehler

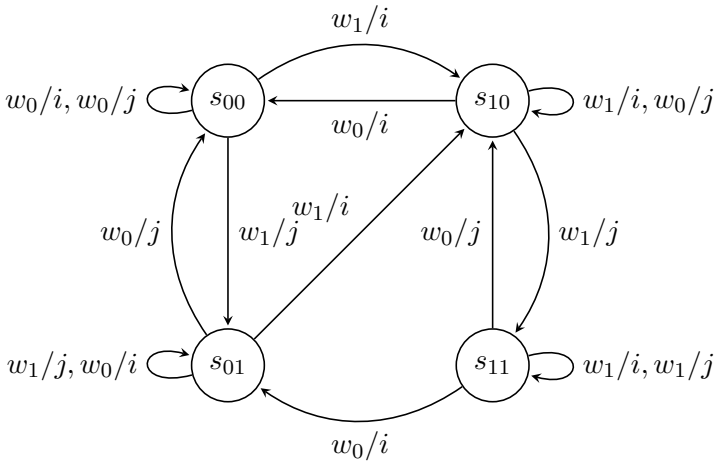


Abbildung 2.5: Speicherzellen mit einem idempotenten Kopplungsfehler

2.4.2 Dynamische Fehler

Speicherfehler können nicht nur statisch sein, sondern können auch von bestimmten Bedingungen abhängig sein, so kann z.B. der gespeicherte Wert einer Zelle auch erst nach einer bestimmten Zeit verfälscht werden oder es wird eine bestimmte Operationsfolge benötigt um den Fehler zu aktivieren. Eine solche Instruktionssequenz wird auch als test unit bezeichnet[ZW06].

Data Retention Fault

Als Datenbewahrungsfehler (engl. Data Retention Fault, DRF) bezeichnet man es, wenn eine Speicherzelle nicht in der Lage ist einen gespeicherten Wert dauerhaft zu halten. Diese Zelle verliert nach einer bestimmten Zeitperiode τ den gespeicherten Wert und nimmt den anderen Wert an. Es existieren zwei verschiedene Typen, zum einen, wenn der Wert `1` nicht dauerhaft gespeichert werden kann und zum anderen, wenn der Wert `0` nicht dauerhaft gespeichert werden kann.

Dynamic Read Destructive Fault (dRDF)

Wenn direkt vor einer Leseoperation auf derselben Speicherzelle bereits eine andere Operation ausgeführt wurde und dadurch der gespeicherte Wert in der Zelle verändert wird und ein falscher Wert gelesen wird, so spricht man von einem dynamischen zerstörenden Lesefehler[HGG03]. Die Operation vor der Leseoperation kann entweder eine Lese- oder Schreiboperation sein, wobei bei diesen zwischen Schreiboperationen unterschieden wird, die den gespeicherten Wert in der Speicherzelle verändern und solchen, die den Wert nicht verändern. Daher wird zwischen drei unterschiedlichen dRDFs unterschieden.

Dynamic Incorrect Read Fault (dIRF)

Wenn direkt vor einer Leseoperation auf derselben Speicherzelle bereits eine andere Operation ausgeführt wurde und dadurch ein falscher Wert gelesen wird, so spricht man von einem dynamischen Lesefehler[HGG03]. Der Inhalt der Speicherzelle bleibt dabei unangetastet. Analog zur Unterscheidung bei den dRDFs unterscheidet man zwischen drei unterschiedlichen dIRFs.

Dynamic Deceptive Read Destructive Fault (dDRDF)

Wenn direkt vor einer Leseoperation auf derselben Speicherzelle bereits eine andere Operation ausgeführt wurde und dadurch der gespeicherte Wert in der Zelle verändert wird, jedoch der korrekte Wert gelesen wird, so spricht man von einem dynamischen trügerischen und zerstörenden Lesefehler[HGG03]. Analog zur Unterscheidung bei den dRDFs unterscheidet man zwischen drei unterschiedlichen dDRDFs.

2.4.3 Fehler im Adressdekoder

Fehler können nicht nur in den Speicherzellen oder der Lese- und Schreiblogik, sondern auch im Adressdekoder auftreten. Man unterscheidet zwischen vier funktionalen Fehlern im Adressdekoder[Goo93][Goo91]:

1. Eine Adresse adressiert keine Speicherzelle
2. Auf eine bestimmte Zelle kann nicht zugegriffen werden
3. Mit einer Adresse werden mehrere Speicherzellen adressiert
4. Auf eine bestimmte Zelle kann von mehreren Adressen aus zugegriffen werden

Da genauso viele Adressen wie Speicherzellen existieren, können diese Fehler nicht allein auftreten, sondern treten immer in einer Kombination auf. Aus diesen vier funktionalen Fehlern des Adressdekoders ergeben sich folgende vier Fehlerkombinationen. Wenn eine Adresse keine Speicherzelle adressiert, so muss entweder auf eine Speicherzelle kein Zugriff möglich sein oder eine andere Adresse muss mehrere Speicherzellen adressieren. Wenn auf eine bestimmte Speicherzelle dagegen von

mehreren Adressen aus zugegriffen werden kann, so kann entweder auf mindestens eine Speicherzelle nicht zugegriffen werden oder eine Adresse muss mehrere Speicherzellen adressieren.

2.5 Marchtests

Es existieren verschiedene Typen von Speichertests. Die Marchtests haben sich dabei als besonders geeignet erwiesen, da die Algorithmen einfach und gut strukturiert sind und außerdem kurze Testdauern erlauben[Goo93].

2.5.1 Definition und Notation

Ein Marchtest besteht aus einer (endlichen) Folge von Marchelementen. Jedes Marchelement besteht aus einer (endlichen) Folge von Speicheroperationen, welche auf einer Zelle ausgeführt wird, bevor diese auf der nächsten Zelle ausgeführt wird. Es gibt vier verschiedene Speicheroperationen:

w0 Eine `0` in eine Zelle schreiben

w1 Eine `1` in eine Zelle schreiben

r0 Lesen, mit Erwartungswert `0`

r1 Lesen, mit Erwartungswert `1`

Zusätzlich existiert zu jedem Element die Information, ob der Speicher aufsteigend ($\Uparrow$) oder absteigend ($\Downarrow$) durchlaufen werden soll oder ob die Richtung frei gewählt werden kann ($\Updownarrow$). Auch wenn der Speicher bei $\Uparrow$ üblicherweise aufsteigend durchlaufen wird, also von Adresse 0 bis Adresse n, so ist das im Prinzip nicht nötig, es reicht aus, dass aufsteigend und absteigend invers zueinander sind, die genaue Reihenfolge kann dabei frei gewählt werden[Goo91]. Daraus folgt auch, dass die Richtungsangaben $\Uparrow$ und $\Downarrow$ in einem Marchtest vertauscht werden dürfen, ohne dass die Fehlerabdeckung dadurch beeinflusst wird.

Für Marchtests hat sich folgende Notation eingebürgert: Der gesamte Marchtest wird mit geschweiften Klammern umschlossen. Innerhalb dieser werden die Marchelemente durch Strichpunkte getrennt. Jedes Marchelement besteht aus der Richtungsangabe, gefolgt von der Operationsfolge, die von Klammern umschlossen wird. Die einzelnen Speicheroperationen der Operationsfolge werden durch Kommata getrennt. Ein Marchtest sieht zum Beispiel so aus: `{⇕(w0);⇕(r0,w1);⇕(r1)}`

2.5.2 Übersicht Marchtests

Im Folgenden werden zunächst einige Marchtests vorgestellt. Anhand der grundlegenden Marchtests MATS, MATS+ und MATS++ wird zunächst die Idee, wie Marchtests konstruiert werden können, erläutert. Der Marchtest March C- deckt alle ausgewählte statische Fehlermodelle ab, während March MD4 die dynamischen Fehlermodelle abdeckt. In Tabelle 2.1 findet sich eine Übersicht der Marchtests und die jeweilige Struktur. Tabelle 2.2 zeigt eine Übersicht über die Fehlerabdeckung, welcher Marchtest welche Fehlermodelle abdeckt.

Name	Notation
MATS[Goo91]	{⇳(w0);⇳(r0,w1);⇳(r1)}
MATS+[Goo91]	{⇳(w0);⇑(r0,w1);⇓(r1,w0)}
MATS++[Goo91]	{⇳(w0);⇑(r0,w1);⇓(r1,w0,r0)}
March C-[Goo91]	{⇳(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇳(r0)}
March MD4[HVZ07]	{⇳(w0,w1,r1,w1,r1,r1,r1,w0,r0,w0,r0,r0,r0)}

Tabelle 2.1: Übersicht Marchtests

	AF	SAF	TF	CFin	CFid	CFst	CFdst	dRDF	dIRF	dDRDF	DRF[1]
MATS	-[2]	+	-	-	-	-	-	-	-	-	-
MATS+	+	+	-	-	-	-	-	-	-	-	-
MATS++	+	+	+	-	-	-	-	-	-	-	-
March C-	+	+	+	+	+	+	+	-	-	-	-
March MD4	-	+	+	-	-	-	-	+	+	+	-

[1] Jeder Test kann erweitert werden, sodass DRF erkannt werden, siehe dazu auch in Kapitel 2.5.3 den Abschnitt über DRF

[2] Bei bekannter Fertigungstechnik möglicherweise Erkennung, siehe dazu [Goo91]

Tabelle 2.2: Übersicht Fehlerabdeckung Marchtests

Zwischen den Fehlerabdeckungen durch Marchtests existieren folgende Zusammenhänge[Goo91]:

- Ein Test für Transitionsfehler erkennt auch Haftfehler
- Ein Test für Kopplungsfehler erkennt auch Transitionsfehler (und damit auch Haftfehler)
- Ein Test für idempotente Kopplungsfehler (CFid) erkennt auch Inversions-Kopplungsfehler (CFin)

2.5.3 Fehlererkennung mit Marchtests

Adressdekoder Fehler

Die vorgestellten Fehlerkombinationen im Adressdekoder können von Marchtests erkannt werden, sofern sie folgende Bedingungen erfüllen[Goo93][Goo91]:

1. $\Uparrow (r_x, \ldots, w_{\overline{x}})$
2. $\Downarrow (r_{\overline{x}}, \ldots, w_x)$

Ein Test, der diese Fehler erkennt, besteht offensichtlich aus mindestens zwei Elementen. Allerdings muss vor der ersten Bedingung der Speicher natürlich mit $x, x \in \{0, 1\}$ initialisiert werden. Innerhalb der Bedingungen kann „…“ durch eine beliebige Folge von Lese- und Schreiboperationen ersetzt

werden. Durch Leseoperationen wird der Speicher nicht verändert und durch die Schreiboperation am Ende des Elements wird sichergestellt, dass am Ende der korrekte Wert im Speicher steht. Außerdem kann am Ende der Elemente eine beliebige Folge von Leseoperationen eingefügt werden, da diese den Speicher nicht verändern. Mit der selben Begründung können zwischen den beiden Bedingungen beliebig viele Leseoperationen eingefügt werden. Der Beweis, dass diese Bedingungen hinreichend und notwendig sind, findet sich in [Goo91].

Haftfehler

Um Haftfehler erkennen und lokalisieren zu können, muss aus jeder Speicherzelle eine `0` und eine `1` gelesen werden[Goo91].

Der einfachste Marchtest, der alle Haftfehler, sowie einen Teil der Fehler im Adressdekoder erkennt, heißt MATS Marchtest (engl. für modified algorithmic test sequence). Listing 2.1 zeigt den Ablauf des Tests. Zunächst wird der gesamte Speicher mit `0` initialisiert. Danach wird im nächsten Marchelement dieser Wert wieder gelesen und der Speicher mit `1` überschrieben. Im letzten Element wird dieser Wert wieder gelesen. Dieser Test erfüllt die oben genannten Bedingungen, jede Speicherzelle wird mit beiden Werten beschrieben und gelesen. Bei n Speicherzellen benötigt der Test $4n$ Operationen. Da nach den Bedingungen jede Zelle zweimal gelesen und geschrieben werden muss, also vier Operationen pro Speicherzelle durchgeführt werden müssen, ist dies ein minimaler Test um Haftfehler zu erkennen.

```
{⇕(w0);⇕(r0,w1);⇕(r1)}
```

Listing 2.1: MATS Marchtest

Um gleichzeitig die Fehler im Adressdekoder erkennen zu können, muss der Test zusätzlich die Bedingungen aus Kapitel 2.5.3 erfüllen. Dazu muss für das zweite und dritte Marchelement die Richtung festgelegt werden und außerdem das dritte Element um eine weitere Schreiboperation ergänzt werden, Listing 2.2 zeigt den daraus folgenden Testablauf. Dieser erweiterte Test heißt MATS+ Marchtest und benötigt $5n$ Operationen. Es werden die in den Bedingungen vorgegebenen vier Operationen, sowie die Initialisierungsoperation ausgeführt, dieser Test ist für die Erkennung der Fehler im Adressdekoder also minimal.

```
{⇕(w0);⇑(r0,w1);⇓(r1,w0)}
```

Listing 2.2: MATS+ Marchtest

Transitionsfehler

Um Transitionsfehler erkennen und lokalisieren zu können, muss jede Zelle sowohl eine `0`→`1` Transition, als auch eine `1`→`0` Transition durchlaufen. Jede Zelle muss außerdem gelesen werden, bevor sie eine weitere Transition durchläuft[Goo91].

Der bereits vorgestellte MATS+ Marchtest (Listing 2.2) erfüllt die erste Bedingung. Im zweiten Marchelement wird eine `0`→`1` Transition durchgeführt und zu Beginn des dritten Marchelements wird jede Zelle zunächst gelesen, bevor sie eine weitere Transition durchläuft. Innerhalb des dritten

Marchelement wird auch bereits eine `1`→`0` Transition durchgeführt, allerdings wird der Wert danach nicht mehr gelesen um zu überprüfen, dass die Transition korrekt durchgeführt wurde. Der Test lässt sich daher sehr einfach erweitern um auch Transitionsfehler zu erkennen. Das letzte Marchelement muss dazu nur um eine weitere Leseoperation am Ende ergänzt werden, die überprüft, ob die Transition korrekt durchgeführt wurde. Der Test erkennt auch nach wie vor Fehler im Adressdekoder, da er die Testbedingungen erfüllt, Leseoperationen am Ende eines Elements einzufügen ist erlaubt. Außerdem werden nach wie vor Haftfehler erkannt, jede Zelle wird nach wie vor zumindest einmal mit jedem Wert geschrieben und gelesen. Dieser erweiterte Test heißt MATS++ Marchtest, Listing 2.3 zeigt den Ablauf. Der Test benötigt folglich $6n$ Operationen. Auch dieser Test ist minimal, da der Speicher zunächst mit einem Wert initialisiert werden muss und diese Initialisierung danach überprüft werden muss. Danach folgen die beiden Transitionen mit jeweils einem Lesevorgang.

```
{⇳(w0);⇑(r0,w1);⇓(r1,w0,r0)}
```

Listing 2.3: MATS++ Marchtest

Kopplungsfehler

Der in Listing 2.4 gezeigte Marchtest March C- erkennt alle vorgestellte statische Fehler, sowie Fehler im Adressdekoder.

Da beide Transitionen zumindest einmal ausgeführt werden, werden Transitionsfehler und damit auch Haftfehler erkannt. Das dritte und vierte Marchelement erfüllen die Bedingung zur Erkennung von Fehlern im Adressdekoder.

Der idempotente Kopplungsfehler, der durch eine `0`→`1`-Transition in der Aggressorzelle ausgelöst wird und eine `0` in der gekoppelten Zelle erzwingt, wird von diesem Test erkannt. Falls die Adresse der gekoppelten Zelle kleiner ist als die der Aggressorzelle, wird der Fehler durch die Leseoperation im dritten Marchelement erkannt, da durch die Transition im zweiten in der gekoppelten Zelle der Wert `0` erzwungen wurde, die Leseoperation im dritten Element jedoch den Wert `1` erwartet. Ist die Adresse der gekoppelten Zelle größer als die der Aggressorzelle, so kann der Fehler nicht im dritten Marchelement erkannt werden, da der erzwungene Wert durch die Schreiboperation bereits wieder überschrieben wurde. Stattdessen wird der Fehler in diesem Fall durch die Leseoperation im fünften Element erkannt, da der Speicher in umgekehrter Richtung durchlaufen wird. Die Erkennung der anderen drei idempotenten Kopplungsfehler erfolgt analog. Da ein Test, der idempotente Kopplungsfehler erkennt, auch Inversions-Kopplungsfehler erkennt, werden diese beiden Fehlermodelle vom March C- Test erkannt.

Der Zustands-Kopplungsfehler, bei dem der Zustand `0` in der Aggressorzelle den Wert `0` in der gekoppelten Zelle erzwingt, wird von diesem Test erkannt. Falls die Adresse der gekoppelten Zelle größer ist als die Adresse der Aggressorzelle, wird dieser Fehler im dritten Marchelement erkannt, da die Schreiboperation `w0` in der Aggressorzelle den Wert `0` in der gekoppelten Zelle erzwingt. Da durch die Leseoperation in diesem Marchelement zunächst eine `1` gelesen werden soll, wird der Fehler erkannt, sobald die gekoppelte Zelle erreicht wird. Falls die Adresse der gekoppelten Zelle kleiner ist als die Adresse der Aggressorzelle, wird der Fehler im fünften Marchelement erkannt, da der Speicher in umgekehrter Richtung durchlaufen wird. Die Erkennung der verbleibenden drei Typen

des Zustand-Kopplungsfehlers erfolgt analog, Zustand-Kopplungsfehler werden also durch den March C- Test erkannt.

Der Störungs-Kopplungsfehler, der durch die Operation `r0` auf der Aggressorzelle ausgelöst wird und die gekoppelte Zelle zu einer `0`→`1`-Transition zwingt, wird von diesem Marchtest erkannt. Falls die Adresse der gekoppelten Zelle größer ist als die der Aggressorzelle, so wird der Fehler im ersten Marchelement erkannt, da in der gekoppelten Zelle dadurch eine `1` gespeichert ist, die Leseoperation dagegen eine `0` erwartet. Falls die Adresse der gekoppelten Zelle kleiner ist als die Adresse der Aggressorzelle, erfolgt die Erkennung im vierten Marchelement, da der Speicher dort in umgekehrter Richtung durchlaufen wird. Die Erkennung der anderen sieben Störungs-Kopplungsfehler erfolgt analog, der Marchtest March C- erkennt also alle Störungs-Kopplungsfehler und damit alle vier Kopplungsfehler.

```
{⇳(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇳(r0)}
```

Listing 2.4: March C- Marchtest

Data Retention Fault

Um einen Data Retention Fault erkennen zu können, muss in jede Zelle zunächst ein Wert geschrieben werden, dann muss die Zeit τ gewartet werden, bevor der Inhalt der Zelle getestet wird, danach muss das selbe mit dem anderen Wert geschehen[Goo93].

Daher kann jeder Marchtest so erweitert werden, dass er diese Fehler erkennt, indem das Schreiben und Lesen in unterschiedlichen Marchelementen geschieht und zwischen der Ausführung dieser Elemente die Zeit τ gewartet wird[GGMY96]: $\{\Updownarrow (\ldots, w_x); \tau; \Updownarrow (r_x, \ldots, w_{\overline{x}}); \tau; \Updownarrow (r_{\overline{x}} \ldots); \}$

Dynamische Lesefehler

Die Aktivierung der dynamischen Lesefehler dRDF, dIRF und dDRDF ist von einer bestimmten Operationsfolge abhängig. Alle drei Fehlermodelle werden durch eine zwei Operationen lange Operationsfolge aktiviert, wobei die zweite Operation eine Leseoperation ist. Die Operation vor der Leseoperation kann dabei, je nach Fehlerart, entweder eine Leseoperation oder eine Schreiboperation sein, wobei bei den Schreiboperationen zwischen den Operationen, die den gespeicherten Wert verändern und denen, die den gespeicherten Wert nicht verändern, unterschieden wird. Der Test muss für beide Werte jeweils alle mögliche Aktivierungssequenzen durchlaufen und den gespeicherten Wert überprüfen. Listing 2.5 zeigt den MD4 Marchtest, der das leistet. Der genaue Beweis dafür und der Beweis, dass dieser Test minimal ist um die Fehlermodelle dRDF, dIRF und dDRDF zu erkennen, findet sich in [HVZ07].

```
{⇳(w0,w1,r1,w1,r1,r1,r1,w0,r0,w0,r0,r0,r0)}
```

Listing 2.5: MD4 Marchtest

2.5.4 Marchtests für wortadressierte Speicher

Die bisher vorgestellten Marchtests gehen davon aus, dass es möglich ist, jede Speicherzelle einzeln zu adressieren, man spricht dabei von bitadressiertem Speicher. Typischerweise sind Speicher jedoch nicht bitadressiert, sondern wortadressiert (mit ≥ 2 Bit pro Wort), man kann also nur ganze Worte (typischerweise 32 oder 64 Bit) adressieren.

Die Marchtests für bitadressierte Speicher können so erweitern werden, dass anstatt einzelner Bits ganze Datenworte zum Testen angelegt werden. Die angelegte Testdaten werden als Testmuster (engl. data backgrounds, DB) bezeichnet. Als zweites Muster wird das invertierte Testmuster genutzt. Es kann jedes beliebige Datenwort als Testmuster gewählt werden, die einzige Bedingung ist, dass als das zweite Muster das invertierte Testmuster gewählt wird. Die Operationen, die bei bitadressierten Speichern eine `0` lesen oder schreiben, lesen oder schreiben bei wortadressierten Speichern das Testmuster ($r_0 \rightarrow r_{\mathrm{DB}}$, $w_0 \rightarrow w_{\mathrm{DB}}$). Die Operationen, die eine `1` lesen oder schreiben, lesen oder schreiben nun das invertierte Testmuster ($r_1 \rightarrow r_{\overline{\mathrm{DB}}}$, $w_1 \rightarrow w_{\overline{\mathrm{DB}}}$). Um die Laufzeit der Tests anzugeben, wird bei einem Speicher mit n Speicherzellen und einer Wortbreite B jedes n durch $\frac{n}{B}$ ersetzt[GT03]. Alternativ kann n auch durch die Anzahl der Worte im Speicher ersetzt werden.

Um die Fehler, die nur eine einzelne Zelle betreffen (SAF, TF, DRF), zu erkennen, genügt diese Umwandlung bereits[GT03]. Ob nur eine Zelle oder mehrere Zellen gleichzeitig gelesen, bzw. geschrieben werden, spielt keine Rolle, da der Wert jeder Zelle unabhängig von allen anderen ist und auch wenn auf mehrere Zellen gleichzeitig zugegriffen wird, wird trotzdem jede einzelne überprüft.

Anders ist das bei Fehlern, die zwei Zellen betreffen. Hierzu muss die bisher eingeführte Fehlerklassifizierung erweitert werden. Man unterscheidet zwischen Kopplungsfehlern, bei denen zwei Zellen gekoppelt sind, die sich innerhalb eines Wortes befinden (engl. intra-word faults) und den Kopplungsfehlern, die zwei Zellen koppeln, die sich in unterschiedlichen Worten befinden (engl. inter-word faults). Letztere entsprechen dabei den Kopplungsfehlern bei bitadressiertem Speicher und werden durch die gleichen Mechanismen wie diese erkannt. Daher genügt für diese Fehler auch eine Erweiterung der Tests, sodass Worte anstatt nur einzelner Bits geschrieben werden.

Kopplungsfehler, die innerhalb eines Wortes auftreten, müssen dagegen gesondert betrachtet werden. Es gibt verschiedene Ansätze, diesen Kopplungsfehlern zu begegnen. Ein grundsätzlicher Ansatz ist es, neue, für wortadressierte Speicher optimierte Marchtests zu entwickeln. Möchte man jedoch auf die bekannten Tests zurückgreifen, so gibt es zwei verschiedene Möglichkeiten. Der traditionelle Ansatz ist, den Test mit unterschiedlichen Testmustern auszuführen[DBT90]. Dazu muss der Test bei B-Bit-Worten $\log_2(B) + 1$ mal ausgeführt werden. Dies führt jedoch zu einer Ausführungszeit von $\log_2(b + 1) \cdot$ *Laufzeit(Marchtest)*. Allerdings werden so weder CFdsts, noch CFids erkannt, sondern nur CFins und CFsts. Ein zielführenderer Ansatz ist einen speziellen Marchtest zu entwickeln, der alle Kopplungsfehler innerhalb eines Wortes abdeckt und diesen dann an den umgewandelten Marchtest anzuhängen[GT03]. Durch eine geschickte Wahl der Testmuster ist es so möglich, mittels $\log_2(B) + 1$ unterschiedlicher Testmuster (zuzüglich ihrer jeweiligen Inversen) und $7 * \log_2(B) + 7$ Speicheroperationen pro Speicherwort alle vier Kopplungsfehler zu entdecken. Bei einem Speicher mit 32 Bit Worten werden also sechs unterschiedliche Testmuster benötigt und es werden 42 Speicheroperatio-

nen durchgeführt, Listing 2.6 zeigt diesen Test. Hängt man diesen Test nun an den umgewandelten Marchtest an, so ergibt sich eine Gesamtlaufzeit von

$$\log_2(B) \cdot \frac{N}{B} + \textit{Laufzeit(Marchtest)}$$

Der Marchtest für die Kopplungsfehler innerhalb eines Wortes kann in eine beliebige Anzahl Marchelemente aufgespalten werden[GT03]. Für jedes Element kann dabei die Richtung, in der der Speicher durchlaufen wird, frei gewählt werden. Außerdem können beliebig viele Leseoperationen eingefügt werden. Dies kann insbesondere dazu genutzt werden, die Testlaufzeit zu reduzieren. Man versucht dabei, Marchelemente zu erzeugen, die bereits im konvertierten Marchtest enthalten sind. Diese müssen für den Marchtest für Kopplungsfehler nicht erneut ausgeführt werden. In [GT03] und [GTH98] wird diese Optimierung anhand von Beispielen erläutert. Dort findet sich auch der Beweis, weshalb dieser Test alle vier Kopplungsfehler innerhalb von Worten erkennt.

DB_0=0x00000000
DB_1=0x55555555
DB_2=0x33333333
DB_3=0x0F0F0F0F
DB_4=0x00FF00FF
DB_5=0x0000FFFF
{⇳(wDB_0, $w\overline{DB_0}$, $r\overline{DB_0}$, $r\overline{DB_0}$, wDB_0, DB_0, rDB_0, wDB_1, $w\overline{DB_1}$, $r\overline{DB_1}$, $r\overline{DB_1}$, wDB_1, rDB_1,
rDB_1, wDB_2, $w\overline{DB_2}$, $r\overline{DB_2}$, $r\overline{DB_2}$, wDB_2, rDB_2, rDB_2, wDB_3, $w\overline{DB_3}$, $r\overline{DB_3}$, $r\overline{DB_3}$, wDB_3,
rDB_3, rDB_3, wDB_4, $w\overline{DB_4}$, $r\overline{DB_4}$, $r\overline{DB_4}$, wDB_4, rDB_4, rDB_4, wDB_5, $w\overline{DB_5}$, $r\overline{DB_5}$, $r\overline{DB_5}$,
wDB_5, rDB_5, rDB_5)}

Listing 2.6: Marchtest für Kopplungsfehler innerhalb eines 32 Bit Wortes mit Testmustern

3 Implementierung

In diesem Kapitel wird das im Rahmen dieser Arbeit implementierte Framework zur Umwandlung von Marchtest-Beschreibungen in softwarebasierte Selbsttestprogramme vorgestellt. Im Anschluss wird die Qualität der erzeugten Selbsttests durch Fehlerinjektion validiert.

3.1 Softwarebasierte Realisierung von Marchtests

Mit dem Framework können Marchtestbeschreibungen mithilfe von Assemblertemplates in softwarebasierte Selbsttests für RISC-Prozessoren umgewandelt werden. Die Templates sind in Assembler für den mimiMips-Prozessor als Beispiel einer RISC-Architektur implementiert. Jedes einzelne Marchelement kann dabei aus bis zu 25 Leseoperationen und beliebig vielen Schreiboperationen bestehen.

Das Framework ist in Python 3 geschrieben und erfordert daher, dass diese Laufzeitumgebung installiert ist. Außerdem muss die Compilersammlung gcc installiert sein, da der Präprozessor genutzt wird.

3.1.1 Testgenerierung

Parameter

Das Framework kann mit verschiedenen Parametern aufgerufen werden, welche im Folgenden detailliert erklärt werden.

```
generateSBST [-h] [-d <DATABACKGROUND>] [-a <ANYDIR> [<ANYDIR> ...]]
             [-b <MEMBASEADR>] [-s <MEMSIZE>] [--endloop] [--intra-word-cf]
             [--stdout] [-o <OUTFILE>] <MARCHFILE>
```

Die in eckigen Klammern angegebenen Parameter sind optional. Die in spitzen Klammern angegebenen Begriffe müssen durch die entsprechenden Übergabewerte ersetzt werden. Im Folgenden werden die Parameter näher erläutert:

Hilfe `-h` oder `--help`
: Zeigt die Hilfe an und liefert eine Erklärung der anderen Parameter.

Testmuster `-d <DATABACKGROUND>` oder `--databackground <DATABACKGROUND>`
Gibt das Datenwort an, das als Testmuster für den Marchtest genutzt werden soll. Dieser Parameter muss in hexadezimaler Darstellung angegeben werden und darf höchstens 8 Stellen haben. Dies entspricht einem 32-Bit-Wort. Wird dieser Parameter nicht angegeben, so wird das Standardtestmuster `0x00000000` genutzt.

Richtung für any `-a <ANYDIR> [<ANYDIR> ...]` oder `--any <ANYDIR> [<ANYDIR> ...]`
Gibt an, wie die Richtungsangabe *any* der Marchelemente interpretiert werden soll. Als Richtungsangabe kann dabei *up* oder *down* übergeben werden. Da in einem Marchtest mehr als ein Marchelement *any* als Richtungsangabe haben kann, können hier beliebig viele Richtungsangaben angegeben werden. Diese werden rückwärts den entsprechenden Marchelementen zu geordnet, das heißt, dem letzten vorkommenden Marchelement mit Richtung *any* wird die letzte übergebene Richtungsangabe zugewiesen, dem vorletzten die vorletzte usw.. Wurden mehr Richtungsangaben übergeben, als entsprechende Marchelemente existieren, so werden die überflüssigen, vorderen Richtungsangaben verworfen. Wurden weniger Richtungsangaben übergeben, so wird die erste für die übrigen Marchelemente genutzt. Wird dieser Parameter nicht angegeben, so wird standardmäßig jedes *any* als *up* interpretiert.

Speicherbasisadresse `-b <MEMBASEADR>` oder `--mem-min-adr <MEMBASEADR>`
Gibt die Basisadresse des zu testenden Speichers an. `<MEMBASEADR>` muss eine 32 Bit-Adresse sein. Wird dieser Parameter nicht angegeben, so wird der Wert 409600 als Basisadresse angenommen.

Speichergröße `-s <MEMSIZE>` oder `--mem-size <MEMSIZE>`
Gibt die Größe des zu testenden Speichers in Worten an. Die maximale Speicheradresse, die sich aus `<MEMBASEADR>` $+ 4 \cdot$ `<MEMSIZE>` ergibt, muss eine gültige 32 Bit Adresse sein. Wird dieser Parameter nicht angegeben, so wird eine Standardgröße von 256 Worten angenommen.

Kopplungsfehler innerhalb eines Wortes `--intra-word-cf`
Wird der Parameter übergeben, so wird der Marchtest für Kopplungsfehler innerhalb eines Wortes an den Test angehängt.

Endlosschleife am Ende `--endloop`
Wird der Parameter übergeben, so wird am Ende des Marchtests eine Endlosschleife erzeugt. Dies kann für Tests und Simulationen genutzt werden.

Ausgabegerät `--stdout`
Wird der Parameter übergeben, so wird der erzeugte Assemblercode auf das Standardausgabegerät geschrieben, anderenfalls wird das Ergebnis in eine Datei gespeichert (`<OUTFILE>` oder `<MARCHFILE>.asm`).

Ausgabedatei `-o <OUTFILE>` oder `--outfile <OUTFILE>`
Die Datei, in die der erzeugte Assemblercode gespeichert werden soll. Wird dieser Parameter nicht angegeben, so wird der Assemblercode in `<MARCHFILE>.asm` gespeichert, anderenfalls in `<OUTFILE>`. Sofern die Ausgabedatei bereits existiert, wird sie ohne Warnung überschrieben.

Datei mit Marchtest `<MARCHFILE>`
Diese Datei enthält die Beschreibung des Marchtests in der Testbeschreibungssprache, diese

wird in Kapitel 3.1.2 erklärt. Enthält diese Datei mehr als einen Marchtest, so wird nur der erste beachtet.

Ablauf der Testgenerierung

1. Parsen der Parameter
   Die Parameter werden zunächst geparst und es wird überprüft, ob sie den in Kapitel 3.1.1 genannten Bedingungen entsprechen.
2. Entfernen der Kommentare aus `<MARCHFILE>`
   Da die Kommentare in der Testbeschreibungssprache im C/C++-Stil gehalten sind, nutzt dieses Framework den C-Präprozessor aus der GNU Compiler-Suite `gcc` um die Kommentare zu entfernen.
3. Parsen des Marchtests
   Der Marchtest wird eingelesen und es wird überprüft, ob er dem in Abschnitt 3.1.2 erklärten Format entspricht. Danach wird er in die interne Datenstruktur überführt.
4. Kopplungsfehler innerhalb eines Wortes
   Sollen die Kopplungsfehler innerhalb eines Wortes erkannt werden, so wird der Marchtest hierfür erzeugt.
5. Assemblercode für Marchelemente
   Der Assemblercode wird für die einzelnen Marchelemente erzeugt.
6. Assemblercode für Initialisierung und Ende des Tests
   Der Assemblercode zur Initialisierung und zum Beenden des Tests wird erzeugt.
7. Speichern des Ergebnisses
   Das erzeugte Assemblerprogramm wird je nach Einstellung auf die Standardausgabe geschrieben oder in einer Datei gespeichert.

Rückgabewerte

Die Ausführung des Frameworks kann mit folgenden Rückgabewerten beendet werden:

0 Erfolgreiche Ausführung. Der Assembler Code wurde erfolgreich erzeugt.

1 Ausführung fehlgeschlagen. Möglicherweise entsprach die Formatierung des Marchtests nicht den Anforderungen oder das Schreiben in die Ausgabedatei ist nicht möglich. Siehe dazu auch die Fehlermeldung in der Standardfehlerausgabe (`stderr`).

2 Ausführung fehlgeschlagen. Die übergebenen Parameter sind fehlerhaft. Siehe dazu auch die Fehlermeldung in der Standardfehlerausgabe (`stderr`).

3.1.2 Testbeschreibungssprache

Zur Beschreibung der Marchtests nutzt das Framework eine Teilmenge der für den Speichersimulator RASTA genutzten Testbeschreibungssprache[BDCDNP02]. Diese erlaubt eine übersichtliche und kompakte Darstellung der Marchtests.

Tabelle 3.1 zeigt eine formale Beschreibung der Testbeschreibungssprache in Form einer EBNF-Grammatik.

START	→	{ *MARCHELEMENT* }
MARCHELEMENT	→	*MARCHELEMENT MARCHELEMENT* |
		<label>:: *DIRECTION* (*OPERATION*);
DIRECTION	→	any | up | down
OPERATION	→	*OPERATION*, *OPERATION* | r0 | r1 | w0 | w1

Tabelle 3.1: Formale Beschreibung des Testbeschreibungssprache

In der Testbeschreibungssprache können Kommentare im Stil der Programmiersprachen C/C++ genutzt werden. Ein Marchtest wird von geschweiften Klammern umschlossen und besteht aus einer Folge von Marchelementen:

```
{
  MARCHELEMENT1
  MARCHELEMENT2
  ...
}
```

Die einzelnen Marchelemente bestehen jeweils aus einem Label, der Richtungsangabe, sowie der Operationsfolge:

```
<label>:: DIR(op1, op2, ...);
```

Dabei gibt `DIR` die Richtung an, in der der Speicher durchlaufen werden soll. Mögliche Werte für `DIR` sind *up* (aufsteigend), *down* (absteigend) oder *any* (aufsteigend oder absteigend). `op1, op2, ...` ist die Operationsfolge, wobei die einzelnen Operationen `op`i, $i \in \{1, 2, \dots\}$ jeweils die Werte *r0*, *r1*, *w0* oder *w1* annehmen können.

Auch wenn es sich hier nicht mehr um einzelne Zellen handelt, sondern Testmuster geschrieben werden, wird hier die Notation wie für einzelne Zellen genutzt. Dabei entsprechen die Operationen $r0$ und $w0$ den Operationen $r\mathrm{DB}$ und $w\mathrm{DB}$ und die Operationen $r1$ und $w1$ den Operationen $r\overline{\mathrm{DB}}$ und $w\overline{\mathrm{DB}}$.

Listing 3.1 zeigt einen Marchtest in der Testbeschreibungssprache. Als Beispiel wurde der in Kapitel 2.5.3 vorgestellte Marchtest MATS++ gewählt: `{⇕(w0);⇑(r0,w1);⇓(r1,w0,r0)}`

```
// Marchtest MATS++
{
m0::  any (w0);
m1::  up (r0, w1);
m2::  down (r1, w0, r0);
}
```

Listing 3.1: MATS++ Marchtest in Testbeschreibungssprache

3.1.3 Registerverwendung im Marchtest

Bei der Ausführung des Marchtests folgt die Verwendung der Prozessorregister folgendem Schema:

$0 Rückgabewert, bei `0` wurde der Speicher getestet und es wurden keine Fehler gefunden, bei `1` wurden Fehler gefunden und bei `-1` wurde der Test nicht komplett ausgeführt oder die Ausführung abgebrochen. Dieses Register wird zu Beginn initialisiert und wird erst am Ende, nach der Testausführung, wieder verändert.

$1 Testmuster, entspricht dem Lesen, bzw. Schreiben von `0` in der Testbeschreibungssprache. Der Wert dieses Registers wird nach der Initialisierung nicht mehr verändert

$2 Bitweise invertiertes Testmuster, entspricht dem Lesen, bzw. Schreiben von `1` in der Testbeschreibungssprache. Der Wert dieses Registers wird nach der Initialisierung nicht mehr verändert.

$3 Speicherbasisadresse, untere Adresse des zu testenden Speicherbereichs. Der Wert dieses Registers wird nach der Initialisierung nicht mehr verändert.

$4 Maximale Speicheradresse, obere Adresse des zu testenden Speicherbereichs. Der Wert dieses Registers wird nach der Initialisierung nicht mehr verändert

$5 Aktuelle Speicheradresse des Speicherworts, das getestet wird. Wird ein Fehler im Speicher gefunden, so steht am Ende des Tests in diesem Register die Adresse des Wortes, in dem der Fehler entdeckt wurde.

$6 - $30 Zwischenspeicherung der Leseergebnisse innerhalb eines Marchelements. $6, $7, $12, $13, $18 und $19 werden während des Marchtests für Kopplungsfehler innerhalb eines Wortes für Testmuster genutzt.

$31 Rücksprungadresse um zum aufrufenden Programm zurückzukehren. Wird während des gesamten Ausführungszeitraums nicht verändert.

3.1.4 Assemblertemplates

Der Assemblercode wird mithilfe von Assemblertemplates erzeugt. Sofern dieses Framework für andere RISC-Architekturen als den miniMIPS genutzt werden soll, so müssen diese Templates angepasst werden. Die Verwendung der Register in den folgenden Templates entspricht dabei der tatsächlichen Verwendung im Framework.

Jeder erzeugte Assembler-Marchtest besteht dabei aus den folgenden Bestandteilen, wobei beliebig viele Marchelemente vorkommen können und der Test für Kopplungsfehler innerhalb eines Wortes optional ist:

- Initialisierung
- Marchelemente
- Kopplungsfehler innerhalb eines Wortes
- Fehlerbehandlung, Ende des Tests

Die zugehörigen Templates werden im Folgenden vorgestellt.

Initialisierung

Bevor der Marchtest selbst durchgeführt werden kann, müssen einige Register initialisiert werden. Listing 3.2 zeigt die komplette Initialisierung. Da ein 32-Bit-Wort nicht am Stück geladen werden kann, werden die beiden Halbworte getrennt geladen. Zunächst wird das Testmuster geladen, das zweite Testmuster ergibt sich dann als Inversion des ersten. Danach wird die untere und die obere Speicheradresse des zu testenden Speichers geladen, bevor schließlich das Ergebnisregister initialisiert wird.

```
; Load Databackground ($1) and inverted Databackground ($2)
lui $1, 43690
addi $1, $1, 43690
nor $2, $1, $1

; Load memory base adress ($3) and memory size ($4)
lui $3, 6
addi $3, $3, 16384
lui $4, 0
addi $4, $4, 256
addi $4, $4, -1
; Byte adressing memory - using memory size for adressing
sll $4, $4, 2
; Load memory max adress ($4)
add $4, $4, $3

; Initialise result register
sub $0, $0, $0
addi $0, $0, -1
```

Listing 3.2: Assemblertemplate Testinitialisierung

Schreiben

Das Template zum Schreiben von Testmustern besteht aus nur einer Instruktion zum Speichern eines Datenworts:

```
sw $1, 0($5)
```

Listing 3.3: Assemblertemplate w0

```
sw $2, 0($5)
```

Listing 3.4: Assemblertemplate w1

Lesen

Die Leseoperation der Marchtests ist bei der Implementierung in Software zweigeteilt. Der im Speicher gespeicherte Wert wird zunächst gelesen, der Vergleich, ob er dem erwarteten Wert entspricht, erfolgt erst danach. Schlägt der Vergleich fehl, so wird die Ausführung des Tests abgebrochen und der Test wird mit einem negativen Rückgabewert beendet.

```
lw $6, 0($5)
bne $1, $6, FAILURE
```

Listing 3.5: Assemblertemplate r0

```
lw $6, 0($5)
bne $2, $6, FAILURE
```

Listing 3.6: Assemblertemplate r1

Die Aktivierung mancher dynamischer Fehler erfordert es, dass Speicheroperationen direkt hintereinander ausgeführt werden. Daher können die beiden Anweisungen des Lesen-Templates im Allgemeinen nicht direkt hintereinander ausgeführt werden. Um die Speicheroperationen direkt hintereinander ausführen zu können darf der Vergleich erst nach allen Speicheroperationen eines Marchelements erfolgen, daher wird für jede Leseoperation innerhalb eines Marchelements ein extra Register benötigt. Da nur eine begrenzte Anzahl an Registern zur Verfügung steht, können mit diesem Framework nicht mehr als 25 Leseoperationen innerhalb eines Marchelements verwendet werden. Ein einfaches Beispiel für ein Marchelement der Form ⇕`(r0,w1,r1)` zeigt Listing 3.7. Es werden zunächst alle Speicheroperationen und die Vergleiche en bloc am Ende durchgeführt.

```
lw $6, 0($5)
sw $2, 0($5)
lw $7, 0($5)
bne $1, $6, FAILURE
bne $2, $7, FAILURE
```

Listing 3.7: Assemblertemplate r0 w1 r1

Marchelement

Ein Marchelement in Assembler besteht aus einem Initialisierungsblock und dem Marchblock. Im Initialisierungsblock wird zunächst die Startadresse für das Marchelement geladen, je nachdem ob der Speicher aufsteigend oder absteigend durchlaufen wird, wird die minimale, bzw. maximale Speicheradresse geladen. Im Marchblock werden zunächst die Speicherzugriffsoperationen des Marchelements ausgeführt. Danach wird überprüft, ob der Speicher bereits komplett durchlaufen wurde, falls ja, so wird zum nächsten Marchelement gesprungen. Anderenfalls wird die Speicheradresse inkrementiert, bzw. dekrementiert und der Marchblock wird erneut ausgeführt.

Der Anfang beider Blöcke wird jeweils mit einem Label markiert. Der Marchblock wird mit dem Label aus der Testbeschreibung eingeleitet, während dem Label des Initialisierungsblocks zusätzlich `init` angehängt wird. Listing 3.8 zeigt das Template eines Marchelements mit dem Label *m0*, das den Speicher aufsteigend durchläuft, Listing 3.9 zeigt dasselbe Marchelement, nur dass der Speicher hier absteigend durchlaufen wird. In beiden Fällen hat das folgende Marchelement das Label *m1*.

```
; Marchelement m0
m0init:
      addi $5, $3, 0
m0:
      [MARCHELEMENTS]
      beq $5, $4, m1init
      addi $5, $5, 4
      j m0
```

Listing 3.8: Assemblertemplate Marchelement (aufsteigend)

```
; Marchelement m0
m0init:
      addi $5, $4, 0
m0:
      [MARCHELEMENTS]
      beq $5, $3, m1init
      addi $5, $5, -4
      j m0
```

Listing 3.9: Assemblertemplate Marchelement (absteigend)

Die einzelnen Marchelemente sind dabei als Schleife implementiert. Eine Implementierung ohne Schleifen würde zwar die Laufzeit des Tests optimieren und eine direkte Hintereinanderausführung der Speicheroperationen ermöglichen, scheidet allerdings aufgrund der Testprogrammgröße und der begrenzten Prozessorressourcen aus. Um einen einfachen Test, wie den MATS-Marchtest, der aus vier Speicheroperationen pro Zelle besteht, auszuführen, bräuchte man bereits mindestens die vierfache Menge des zu testenden Speichers nur um das Testprogramm zu speichern. Da Prozessoren sehr viel weniger Register als Speicher besitzen, ist es außerdem nicht möglich sämtliche gelesene Daten gleichzeitig zu speichern und am Ende zu vergleichen. Die Ausführung aller Speicheroperationen in einem Speicherdurchlauf ist nicht am Stück möglich, daher wird beim Durchlaufen des Speichers nach

jeder Adresse zunächst überprüft, ob die korrekten Werte gelesen wurden, bevor das Marchelement auf der nächsten Adresse angewendet wird.

Kopplungsfehler innerhalb von Worten

Der Test für Kopplungsfehler innerhalb eines Wortes benötigt bei einer Wortbreite von 32 Bit sechs unterschiedliche Testmuster (siehe auch Kapitel 2.5.4). Um alle sechs Testmuster (und ihre jeweiligen Inverse) in einem Marchelement schreiben und jeweils zwei mal lesen zu können, wären mindestens 52 Register nötig, zuzüglich der Register für den Programmablauf (mindestens 3 für die Speicheradressen). Da der miniMIPS jedoch nur 32 Register zur Verfügung stellt, wird ausgenutzt, dass der Test für Kopplungsfehler in mehrere Marchelemente aufgeteilt werden kann (vgl. Kapitel 2.5.4). Der Test wird in zwei Marchelemente aufgeteilt und aus Performancegründen als Template umgesetzt.

Testende

Das Template für das Ende der Marchtests beinhaltet die Fehlerbehandlung, falls der Test fehlgeschlagen ist und setzt den Rückgabewert in Register `$0`.

```
FAILURE:
      addi $0, $0, 2
      j END
DONE:
      addi $0, $0, 1
END:
```

Listing 3.10: Assemblertemplate Testende

Falls am Ende eine Endlosschleife erzeugt werden soll, so wird am Ende noch die Anweisung `j END` angefügt.

3.2 Validierung der Fehlerabdeckung mittels Fehlerinjektion

Da die softwarebasierte Durchführung von Marchtests bei bestimmten dynamischen Fehlermodellen zu geringerer Fehlerabdeckung führen kann, wird die Qualität der erzeugten Selbsttestprogramme experimentell durch Simulation und Fehlerinjektion ermittelt. Um die Fehlerinjektion zu ermöglichen, wird der simulierte Speicher um eine Schnittstelle zur Fehlerinjektion erweitert.

3.2.1 Fehlermodelle

Dem erweiterten Speicher können dynamische Lesefehler injiziert werden, die durch eine vorherige Operation aktiviert werden müssen. Die injizierten Lesefehler können sich auf drei verschiedene Arten auswirken. Zunächst indem die Ausgabe bei der Leseoperation invertiert wird und das Speicherwort

erhalten bleibt. Eine zweite Möglichkeit ist, dass die Ausgabe den korrekten Wert ausgibt, das Speicherwort jedoch invertiert wird. Die dritte ist, dass die Ausgabe und zusätzlich auch die Speicherzelle invertiert wird. Die aktivierenden Operationen vor der Leseoperation kann eine Leseoperation, eine Schreiboperation, die den Zelleninhalt verändert, oder eine Schreiboperation, die den Zelleninhalt nicht verändert, sein. Diese Fehler entsprechen den drei in Kapitel 2.4.2 vorgestellten Fehlermodellen dRDF, dIRF und dDRDF. Die drei Fehlerarten, bei denen sowohl der gespeicherte Wert, als auch die Ausgabe invertiert werden, sind die drei unterschiedlichen Fehlerarten des Modells dRDF. Die drei Fehlerarten, bei denen nur die Ausgabe invertiert wird, sind die drei unterschiedlichen Fehlerarten des Modells dIRF und die verbleibenden drei Fehlerarten, bei denen nur der gespeicherte Wert invertiert wird, stellen die drei unterschiedlichen Fehlerarten des Modells dDRDF dar.

3.2.2 Schnittstelle

Um die Fehler in den erweiterten Speicher injizieren zu können, wurde die Schnittstelle des Speichers um die in Tabelle 3.2 vorgestellten Eingänge erweitert. Der Adresseingang `fi_adr` gibt die Adresse an, an der der Fehler injiziert wird. Mit den beiden Eingängen `fi_r_w` und `fi_w_trans` wird festgelegt, welche Operation den Fehler aktiviert. Dabei wird mit dem Eingang `fi_r_w` zunächst festgelegt, ob die aktivierende Operation eine Lese- (`0`) oder Schreiboperation (`1`) ist. Sofern es sich um eine Schreiboperation handelt, wird mit `fi_w_trans` festgelegt, ob es sich bei der aktivierenden Operation um eine den gespeicherten Wert verändernde oder eine den gespeicherten Wert nicht verändernde Schreiboperation handelt. Die Eingänge `fi_cell_flip` und `fi_out_flip` werden dazu genutzt um das Fehlermodell auszuwählen, aus dem Fehler injiziert werden sollen. Werden beide Eingänge auf `0` gesetzt, so wird kein Fehler injiziert. Wird nur `fi_out_flip` auf `1` gesetzt, so wird die Ausgabe invertiert. Wird nur `fi_cell_flip` auf `1` gesetzt, so wird nur der gespeicherte Wert invertiert. Werden beide auf `1` gesetzt, wird sowohl die Ausgabe, als auch der gespeicherte Wert invertiert. Tabelle 3.3 zeigt die Zuordnung der Fehlermodelle zu den Eingangskombinationen.

Name	Typ	Funktion
`fi_adr`	`std_logic_vector(31 downto 0)`	Adresse, an der der Fehler injiziert wird
`fi_r_w`	`std_logic`	Bei `1` ist die aktivierende Operation eine Schreiboperation, anderenfalls eine Leseoperation
`fi_w_trans`	`std_logic`	Typ der aktivierenden Schreiboperation. Bei `1` verändernde Schreiboperation, anderenfalls nicht verändernde Schreiboperation, bei aktivierender Leseoperation irrelevant
`fi_cell_flip`	`std_logic`	Bei `1` wird der gespeicherte Wert invertiert, anderenfalls nicht
`fi_out_flip`	`std_logic`	Bei `1` wird die Ausgabe invertiert, anderenfalls nicht

Tabelle 3.2: Schnittstelle zur Fehlerinjektion

Fehlermodell	`fi_cell_flip`	`fi_out_flip`
Fehlerfrei	`0`	`0`
dRDF	`1`	`1`
dIRF	`0`	`1`
dDRDF	`1`	`0`

Tabelle 3.3: Übersicht Fehlermodelle und Fehlerinjektion

4 Ergebnisse

Zunächst werden die Ergebnisse der Untersuchung der Testlaufzeit vorgestellt. Danach wird die experimentelle Bestimmung der Fehlerabdeckung erklärt. Schließlich werden die Ergebnisse der Bestimmung der Fehlerabdeckung präsentiert.

4.1 Testdauer

Die Ausführungsdauer der softwarebasierten Marchtests wächst, wie bei der herkömmlichen Implementierung von Marchtests, linear mit der Anzahl der Speicherzellen. Tabelle 4.1 zeigt die Anzahl der Befehle, die ausgeführt werden müssen um die ausgewählten Marchtests softwarebasiert auszuführen. Zusätzlich zu den Speicheroperationen muss noch eine erhebliche Anzahl weiterer Anweisungen ausgeführt werden. Dies liegt zum einen daran, dass die Marchoperation, die gleichzeitig den Wert liest und überprüft, in Software in zwei Befehle aufgeteilt werden muss. Zum anderen liegt dies am Overhead, der durch die Implementierung der Marchelemente als Schleife entsteht.

Während die Ausführungsdauer der Marchtests bei herkömmlichen Testverfahren nur von der Anzahl der Speicheroperationen k und der Speichergröße n abhängig ist, spielen bei der Implementierung als softwarebasierter Selbsttest weitere Parameter eine Rolle. Da das Lesen und Validieren des gelesenen Wertes in Software nicht in einer Operation möglich ist, sondern in zwei Befehlen geschehen muss, muss zwischen der Anzahl der Leseoperationen r und der Anzahl der Schreiboperationen w unterschieden werden (wobei gelten muss: $r + w = k$). Da die einzelnen Marchelemente als Schleife

Marchtest	Länge[1]	Befehle (gesamt)	ALU-Befehle[2]	Sprungbefehle[3]	Speicherbefehle[4]
MATS	$4n$	$15n + 10$	$3n + 13$	$8n - 3$	$4n$
MATS+	$5n$	$16n + 10$	$3n + 13$	$8n - 3$	$5n$
MATS++	$6n$	$18n + 10$	$3n + 13$	$9n - 3$	$6n$
March C-	$10n$	$33n + 7$	$6n + 13$	$17n - 6$	$10n$
March MD4	$13n$	$24n + 12$	$n + 13$	$10n - 1$	$13n$

[1] Anzahl Testoperationen des Marchtests
[2] ALU- oder ALU-ähnliche Befehle, hier: `addi`, `lui`, `nor`, `sll`, `sub`
[3] Sprungbefehle, hier: `beq`, `bne`, `j`
[4] Befehle, die auf den Speicher zugreifen, hier: `lw`, `sw`

Tabelle 4.1: Effizienz SBST-Marchtests

implementiert sind, entstehen in jedem Marchelement einige Anweisungen Overhead für die Schleifensteuerung und das Anpassen der Speicheradresse. Dadurch wird die Anzahl der Marchelemente e für die Laufzeit des Tests relevant. Des Weiteren müssen für die Initialisierung und das Beenden des Tests einige Operation am Anfang, bzw. am Ende des Tests durchgeführt werden. Die Länge des Tests lässt sich mit folgender Formel berechnen:

$$\text{Länge}(n, r, w, e) = (w + 2r)n + (3n - 1)e + 13$$

Die Formel setzt sich aus folgenden Bestandteilen zusammen:

- Initialisierung des Tests: 12 Anweisungen
- Schreiboperationen: $w \cdot n$ Anweisungen
- Leseoperationen (inklusive Wertvergleich): $2r \cdot n$ Anweisungen
- Initialisierung der Marchelemente: e Anweisungen
- Overhead der Marchelemente beim Durchlaufen des Speichers: $(3n - 2)e$ Anweisungen
- Beenden des Tests: 1 Anweisung

Anhand dieser Formel zeigt sich, dass, im Gegensatz zu Hardwaretestverfahren, die Anzahl der Marchelemente für die Testlänge bei der Implementierung als softwarebasierter Selbsttest eine sehr große Rolle spielt. Verdeutlichen lässt sich der große Einfluss der Anzahl der Marchelemente am Marchtest MD4, von dem es mehrere Varianten gibt, die alle die dynamischen Lesefehler abdecken. Tabelle 4.2 zeigt zwei Varianten, eine mit einem und eine mit sieben Marchelementen, wobei für beide pro Speicheradresse 13 Speicheroperationen ausgeführt werden müssen. Während für die Variante mit einem Marchelement insgesamt $24n + 12$ Anweisungen benötigt werden müssen, müssen für die Variante mit sieben Marchelementen insgesamt $42n + 6$ Anweisungen ausgeführt werden.

Da ALU-, Sprung- und Speicherbefehle eine unterschiedliche Ausführungsdauer haben, zeigt Tabelle 4.1 zusätzlich die Anzahl der Befehle nach diesen Befehlsgruppen aufgeschlüsselt. Berechnet werden kann die Anzahl der jeweiligen Befehle mit den folgenden Formeln:

$$\begin{aligned} \#\text{ALUOPs} &= 13 + e \cdot n \\ \#\text{SprungOPs} &= (2e + r)n - e \\ \#\text{SpeicherOPs} &= w + r = k \end{aligned}$$

Da bei softwarebasierten Tests die Testprogramme in den Speicher übertragen oder sogar im ROM gespeichert werden müssen, ist spielt die Größe der erzeugten Testprogramme eine große Rolle.

Marchelemente	Notation	Länge	Befehle
$e = 1$	{⇕(w0,w1,r1,w1,r1,r1,r1,w0,r0,w0,r0,r0,r0)}	$13n$	$24n + 12$
$e = 7$	{⇕(w0);⇕(w1,r1);⇕(w1,r1,r1);⇕(r1); ⇕(w0,r0);⇕(w0,r0,r0);⇕(r0)}	$13n$	$42n + 6$

Tabelle 4.2: Vergleich unterschiedliche MD4-Marchtests

Marchtest	Länge	Programmgröße (# Befehle)	Programmgröße (Byte)
MATS	$4n$	33	132 Byte
MATS+	$5n$	34	136 Byte
MATS++	$6n$	36	144 Byte
March C-	$10n$	48	192 Byte
March MD4	$13n$	39	152 Byte

Tabelle 4.3: Testprogrammgröße SBST-Marchtests

Tabelle 4.3 zeigt den Speicherbedarf der einzelnen Tests als Testprogramm. Auch hier zeigt sich, dass die Programmgröße nicht nur von der Anzahl der Speicheroperationen, sondern insbesondere von der Anzahl der Marchelemente abhängig ist.

4.2 Fehlerabdeckung

Der zweite Aspekt der Untersuchung beschäftigt sich mit der Frage, ob und inwieweit die Fehlerabdeckung der Marchtests durch die Implementierung als softwarebasierter Selbsttest beeinträchtigt wird.

Da die Aktivierung der dynamischen Lesefehler von einer ununterbrochenen Aktivierungssequenz abhängig ist, wird die Fehlerabdeckung dieser Fehlermodelle experimentell durch Fehlerinjektion bestimmt. Dazu wird zunächst der Marchtest in der Testbeschreibungssprache beschrieben und mithilfe des vorgestellten Frameworks in Assemblercode für den miniMIPS-Prozessor umgewandelt. Dieser wird mit dem zum miniMIPS gehörenden Assembler in Maschinencode kompiliert. Die Ausführung dieses Codes wird dann mit Modelsim simuliert. Durch die entsprechende Ansteuerung des erweiterten Speichers werden dann die einzelnen Fehler injiziert. Die Testergebnisse werden gespeichert und dann validiert. Zur Simulation wird ein Speicher mit einer Speicherkapazität von 256 Worten, (1 Kilobyte) genutzt. Jeder Fehler wird dabei an jeder Adresse injiziert, bei Marchelementen mit der Richtungsangabe *any* wird immer die Richtung *up* angenommen.

4.2.1 Statische Fehler

Die Erkennung statischer Fehler ist nicht von speziellen Aktivierungssequenzen oder bestimmten Zeitverhalten abhängig, sondern nur davon, dass alle Speicherzellen bestimmte Zustände erreicht oder bestimmte Transitionen durchlaufen haben. Der genaue Zeitpunkt der Überprüfung der Daten ist also nicht relevant, mögliche Verzögerungen durch das Implementieren in Software beeinflussen die Fehlerabdeckung nicht. Beim Testen statischer Fehler mittels softwarebasierter Marchtests entsteht keine Verringerung der Fehlerabdeckung. Ein Marchtest, der statische Fehler erkennt, erreicht bei der softwarebasierten Umsetzung für diese Fehler die gleiche Fehlerabdeckung wie bei der Umsetzung in Hardware.

4.2.2 Dynamische Fehler

Das Wesen der dynamischen Fehler erfordert dagegen, dass Timings eingehalten oder bestimmte Aktivierungssequenzen ausgeführt werden. Die Fehlerabdeckung kann daher dadurch beeinflusst werden, ob bestimmte Speicheroperationen direkt hintereinander ausgeführt werden können oder nicht.

4.2.3 Data Retention Fault

Auch die Erkennung von DRF-Fehlern ist mit softwarebasierten Selbsttests möglich. Die Erweiterung der Marchtests (Kapitel 2.5.3) um auf diese Fehler zu testen erfordert, dass zwischen dem Schreiben und Lesen eine bestimmte Mindestzeit gewartet wird. Diese Erweiterung ist z.B. durch eine Schleife, die einen Zähler entsprechend lange hoch zählt, auch in Software problemlos umsetzbar.

4.2.4 Dynamische Lesefehler

Tabelle 4.4 zeigt die Übersicht, welcher Marchtest welche dynamische Lesefehler, aufgeschlüsselt nach den aktivierenden Operationen, erkennen. Die Marchtests MATS, MATS+ und C- erkennen, wie auch bei der herkömmlichen Implementierung, konstruktionsbedingt keinen der dynamischen Lesefehler, da innerhalb der Marchelemente keine Leseoperation direkt auf eine Schreiboperation folgt. Der MATS++ Marchtest enthält eine Leseoperation direkt nach einer Schreiboperation, die den Zelleninhalt verändert. Daher kann dieser Test die Fehlerarten, die durch diese Operation aktiviert werden und direkt die Ausgabe verändern, erkennen. Diese Fehlerabdeckung entspricht der Fehlerabdeckung bei der herkömmlichen Implementierung.

Der Marchtest MD4, der für die Erkennung dieser dynamischen Lesefehler konstruiert ist, erkennt auch als softwarebasierter Selbsttest alle neun Fehler der drei dynamischen Lesefehlermodelle. Durch die Implementierung der Marchelemente, sodass die Speicheroperationen innerhalb eines Marchelementes direkt hintereinander ausgeführt werden können, wird die erforderte Hintereinanderausführung der Operationsfolge, die die Fehler aktiviert, sichergestellt.

Test	dRDF			dIRF			dDRDF		
	read	write_nt[1]	write_t[2]	read	write_nt[1]	write_t[2]	read	write_nt[1]	write_t[2]
MATS	0%	0%	0%	0%	0%	0%	0%	0%	0%
MATS+	0%	0%	0,4%	0%	0%	0,4%	0%	0%	0%
MATS++	0%	0%	100%	0%	0%	100%	0%	0%	0%
C-	0%	0%	0,8%	0%	0%	0,8%	0%	0%	0%
MD4	100%	100%	100%	100%	100%	100%	100%	100%	100%

[1] Schreiboperation, bei der der gespeicherte Wert nicht verändert wird (engl. non transition write)
[2] Schreiboperation, bei der der gespeicherte Wert verändert wird (engl. transition write)

Tabelle 4.4: Fehlerabdeckung bei Fehlerinjektion

5 Fazit

Trotz der funktionalen Nebenbedingungen bei der Implementierung von Marchtests als softwarebasierte Selbsttests wächst die Laufzeit der Tests nach wie vor linear mit der Anzahl der Speicherzellen. Durch zusätzliche Anweisungen für Kontrollstrukturen wird die Testdauer der softwarebasierten Marchtests zu einem erheblichen Anteil durch die Anzahl der Marchelemente des Marchtests bestimmt.

Die Fehlerabdeckung der statischen Fehlermodelle mittels der vorgestellten Marchtests wird durch die softwarebasierte Testimplementierung nicht beeinträchtigt. Es konnte experimentell gezeigt werden, dass auch die Fehlerabdeckung der dynamischen Lesefehler dRDF, dIRF und dDRDF durch die Implementierung in Software nicht beeinträchtigt wird.

Der vorgestellte Ansatz schränkt die Fehlerabdeckung für statische, sowie dynamische Fehlermodelle, die durch minimale Timingbedingungen oder durch Aktivierungssequenzen innerhalb eines Marchelements aktiviert werden, nicht ein. Wird die direkte Hintereinanderausführung von Anweisungen über mehrere Adressen erforderlich, so scheint die Implementierung als softwarebasierter Selbsttest an ihre Grenzen zu stoßen, da der ganze Speicher aufgrund der geringen Registerzahl und des Speicherbedarfs eines solchen Testprogramms nicht am Stück durchlaufen werden kann.

6 Ausblick

Um die Testdauer der softwarebasierten Marchtests zu optimieren sind zwei Ansätze denkbar.

Zum einen ist es möglich die Marchtests selbst zu optimieren. Bei einigen Marchtests ist es möglich bestimmte Marchelemente zusammenzufassen, bzw. zu trennen. Für die Implementierung als softwarebasierter Marchtest sollten diese aus so wenig Marchelementen wie möglich bestehen, da jedes Marchelement in Software zusätzliche Anweisungen als Kontrollstrukturen benötigt.

Zum anderen kann untersucht werden, inwieweit ein teilweises Entrollen der Marchelementschleifen, sodass mehrere Speicheradressen in einem Schleifendurchlauf getestet werden, einen Optimierungsansatz darstellt. Dadurch wird die Laufzeit des Testprogramms verkürzt, da die Schleife seltener durchlaufen werden muss. Dabei wird jedoch die Anzahl der möglichen Leseoperationen innerhalb eines Marchelements reduziert und insbesondere die Programmgröße vergrößert.

Außerdem sind Untersuchungen bezüglich der softwarebasierten Umsetzung von Marchtests für weitere dynamische Fehlermodelle nötig. Dies umfasst sowohl Fehlermodelle, die nur in begrenzten Zeitfenstern aktiv sind, als auch solche, die durch eine Sequenz von Zugriffen über mehrere Speicheradressen aktiviert werden.

Literaturverzeichnis

[AGP+09] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, M. S. Reorda. Test program generation for communication peripherals in processor-based SoC devices. *IEEE Design & Test of Computers*, 26(2):52–63, 2009. (Zitiert auf Seite 9)

[BA00] M. Bushnell, V. D. Agrawal. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*, Band 17. Springer Science & Business Media, 2000. (Zitiert auf den Seiten 9, 11 und 13)

[BDCDNP02] A. Benso, S. Di Carlo, G. Di Natale, P. E. Prinetto. Specification and design of a new memory fault simulator. In *Proceedings of the Asian Test Symposium (ATS).*, S. 92–97. IEEE Computer Society, Los Alamitos (CA), 2002. doi:10.1109/ATS.2002.1181693. (Zitiert auf Seite 30)

[DBT90] R. Dekker, F. Beenker, L. Thijssen. A realistic fault model and test algorithms for static random access memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):567–572, 1990. (Zitiert auf Seite 25)

[GGH10] A. van de Goor, G. Gaydadjiev, S. Hamdioui. Memory testing with a RISC microcontroller. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).*, S. 214–219. European Design and Automation Association, 2010. (Zitiert auf Seite 13)

[GGMY96] A. van de Goor, G. Gaydadjiev, V. Mikitjuk, V. Yarmolik. March LR: a test for realistic linked faults. In *Proceedings of VLSI Test Symposium (VTS).*, S. 272–280. 1996. doi:10.1109/VTEST.1996.510868. (Zitiert auf den Seiten 17 und 24)

[GHS+12] M. Grosso, W. P. Holguin, E. Sánchez, M. S. Reorda, A. Tonda, J. V. Medina. Software-Based Testing for System Peripherals. *Journal of Electronic Testing*, 28(2):189–200, 2012. (Zitiert auf Seite 9)

[Goo91] A. J. Van de Goor. *Testing semiconductor memories: theory and practice.* John Wiley & Sons, Inc., 1991. (Zitiert auf den Seiten 9, 13, 14, 15, 17, 19, 20, 21 und 22)

[Goo93] A. van de Goor. Using march tests to test SRAMs. *IEEE Design & Test of Computers*, 10(1):8–14, 1993. doi:10.1109/54.199799. (Zitiert auf den Seiten 19, 20, 21 und 24)

[GT03] A. J. Van de Goor, I. B. Tlili. A systematic method for modifying march tests for bit-oriented memories into tests for word-oriented memories. *IEEE Transactions on Computers*, 52(10):1320–1331, 2003. (Zitiert auf den Seiten 25 und 26)

[GTH98] A. van de Goor, I. Tlili, S. Hamdioui. Converting March tests for bit-oriented memories into tests for word-oriented memories. In *Proceedings of the International Workshop on Memory Technology, Design and Testing (MTDT)*., S. 46–52. 1998. doi:10.1109/MTDT.1998.705945. (Zitiert auf Seite 26)

[HGG03] S. Hamdioui, G. N. Gaydadjiev, A. J. van de Goor. A Fault Primitive Based Analysis of Dynamic Memory Faults. In *IEEE Annual Workshop on Circuits, Systems and Signal Processing, Veldhoven, the Netherlands*, S. 84–89. 2003. (Zitiert auf Seite 19)

[HVZ07] G. Harutunyan, V. Vardanian, Y. Zorian. Minimal March Tests for Detection of Dynamic Faults in Random Access Memories. *Journal of Electronic Testing*, 23(1):55–74, 2007. doi:10.1007/s10836-006-9504-8. (Zitiert auf den Seiten 21 und 24)

[PGH+06] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, S. Ravi. Systematic software-based self-test for pipelined processors. In *Proceedings of the annual Design Automation Conference (DAC)*., S. 393–398. ACM, 2006. (Zitiert auf Seite 9)

[PGSR10] M. Psarakis, D. Gizopoulos, E. Sanchez, M. S. Reorda. Microprocessor software-based self-testing. *Memory*, 10(111):01111, 2010. (Zitiert auf den Seiten 9 und 12)

[Raj99] R. Rajsuman. Testing a system-on-a-chip with embedded microprocessor. In *Proceedings of the International Test Conference (ITC)*., S. 499–508. IEEE, 1999. (Zitiert auf Seite 13)

[RGT+14] F. Reimann, M. Glaß, J. Teich, A. Cook, L. R. Gómez, D. Ull, H.-J. Wunderlich, U. Abelein, P. Engelke. Advanced diagnosis: SBST and BIST integration in automotive E/E architectures. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*., S. 1–6. IEEE, 2014. (Zitiert auf Seite 13)

[TKPG10] G. Theodorou, N. Kranitis, A. Paschalis, D. Gizopoulos. A software-based self-test methodology for in-system testing of processor cache tag arrays. In *IEEE International On-Line Testing Symposium (IOLTS)*., S. 159–164. IEEE, 2010. (Zitiert auf den Seiten 9 und 13)

[TKPG11] G. Theodorou, N. Kranitis, A. Paschalis, D. Gizopoulos. A software-based self-test methodology for on-line testing of processor caches. In *IEEE International Test Conference (ITC)*., S. 1–10. IEEE, 2011. (Zitiert auf den Seiten 9 und 13)

[Wun09] H.-J. Wunderlich. *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*, Band 43. Springer Science & Business Media, 2009. (Zitiert auf den Seiten 11 und 15)

[ZW06] J. Zhou, H.-J. Wunderlich. Software-based self-test of processors under power constraints. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*., S. 430–435. European Design and Automation Association, 2006. (Zitiert auf Seite 18)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift