Effiziente Auswahl redundanter Komponenten für Prozessoren zur Kompensation permanenter Fehler

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus – Senftenberg

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Master of Science (M. Sc.)
Tobias Koal
Geboren am 3.1.1981 in Cottbus

Gutachter: Prof. Dr. M. Gössel
Gutachter: A.o. Univ.-Prof. Dr. A. Steininger
Gutachter: Prof. Dr. H. T. Vierhaus

Tag der mündlichen Prüfung: 2014/07/03

Danksagung

Herrn Prof. Dr. H.T. Vierhaus möchte ich recht herzlich für die Möglichkeit der Ausarbeitung dieser Dissertation und die Bereitstellung des dafür notwendigen Arbeitsplatzes danken. Darüber hinaus bedanke ich mich bei ihm für die Betreuung über den gesamten Zeitraum der Ausarbeitung dieser Arbeit. Weiterhin war die gewährte Freiheit bei Untersuchung und Entwicklung ein ebenso wichtiger Punkt wie die vielen wertvollen Diskussionen und sein vorhandenes Wissen in den angrenzenden Fachgebieten.

Ebenso möchte ich mich bei Herrn A.o. Univ.-Prof. Dr. A. Steininger und Herrn Prof. Dr. M. Gössel für die Begutachtung dieser Dissertationsschrift sowie die daraus hervorgegangenen wertvollen Hinweise bedanken.

Ein ganz besonderer Dank geht an Dr. Mario Schölzel, der stets ein wertvoller Ansprechpartner war und diese Arbeit durch seine Ideen, Anregungen und konstruktive Kritik bereicherte. Danke für diese sehr gute Zusammenarbeit in den letzten Jahren.

Weiterer Dank geht auch an alle Mitarbeiter (Christian, Sebastian, Markus, Roberto, Uwe, Stefan, Daniel und natürlich Kathleen) vom Lehrstuhl Technische Informatik der BTU Cottbus – Senftenberg die mich über die gesamte Zeit immer wieder auf ganz unterschiedliche Weise motiviert und unterstützt haben.

Allen meinen Freunden und meiner gesamten Familie danke ich für die umfangreiche Unterstützung vor, während und hoffentlich auch nach der Vollendung dieser Arbeit. Vielen Dank, dass ihr mich ab und an gedanklich befreien konntet.

Besonders möchte ich hier Katja und Christian für das Finden der von mir liebevoll versteckten Rechtschreibfehler danken. Ihr habt sie fast alle gefunden!

An dieser Stelle folgt der Dank an die wichtigste Person meines Lebens, meine Frau Marlen. Ich weiß, dass diese Zeit sehr anspruchsvoll für uns beide war. Ich bin froh, dass es vorbei ist und bin mir sicher du bist es auch. Vielen Dank für deine Geduld und Nachsichtigkeit während dieser schwierigen Zeit.

Zusammenfassung

Die stetige Skalierung von Fertigungstechnologien sorgte für einen rasanten Anstieg der Komplexität und damit auch der Verarbeitungsleistung von integrierten Schaltungen. Dies führte auch zu höheren Anforderungen an die Entwurfs- und Produktionsprozesse für diese Systeme. Zusätzlich dazu steigern Strukturen im Nanometerbereich die Anfälligkeit gegenüber physikalischen Effekten, welche sich in temporären und zunehmend auch dauerhaften Störungen der Funktionalität äußern können. Der Einsatz von Fehlertoleranz ist für diese komplexen Systeme nicht wegzudenken und wird für zukünftige anfälligere Fertigungstechnologien noch relevanter.

In dieser Arbeit wird eine skalierbare Architektur zur Kompensation dauerhafter Störungen für beliebige Prozessorkomponenten vorgestellt. Der Einsatz dieser Architektur ist unabhängig von der Fehlerursache und kann sowohl direkt nach der Produktion als auch während des Einsatzes im Zielsystem genutzt werden. Durch die Verwendung dieser Architektur, auf aktiver Hardware-Redundanz basierend, ist eine Steigerung der Zuverlässigkeit, der Lebensdauer aber auch der Produktionsausbeute bei gleichbleibender Funktionalität möglich. Mit der Modellierung in dieser Arbeit wird die Effizienz der vorgestellten Architektur, unter Berücksichtigung der zusätzlichen Hardware für Redundanz und der notwendigen administrativen Komponenten, ermittelt und ermöglicht damit einen zielgerichteten Auswahlprozess für Prozessorkomponenten und die Menge ihrer Redundanz. Somit wird die optimale Redundanz für ein gegebenes System und ein zu erreichendes Ziel bereits im Entwurfsprozess bestimmt und kann damit frühzeitig bei der Umsetzung berücksichtigt werden.

Neben der Beschreibung des Aufbaus der Architektur und ihrer Funktionsweise zeigt diese Arbeit wie sich eine Integration in bestehende Entwurfsprozesse mit gängigen Methoden und Werkzeugen realisieren lässt. Zusätzlich dazu wird die Systemmodellierung zur Realisierung des zielgerichteten Auswahlprozesses beschrieben. Anhand eines Anwendungsbeispiels werden die Möglichkeit der Umsetzung aufgezeigt und die daraus resultierenden Ergebnisse diskutiert.

Abstract

Steadily downscaling of production technologies led to a rapid increase in complexity and computing power of integrated circuits. This development raises also the requirements of design- and production processes of those systems. Structures in the nanometer regime enhance the susceptibility against physical effects, which can cause temporal and evermore also permanent faults. The usage of fault tolerance became essential for those complex systems and will be even more crucial in future technologies.

This thesis presents a scalable hardware architecture for permanent fault compensation in arbitrary processor components. The utilization of this architecture is independent to the fault cause and is therefore suitable for fault compensation after production as well as in the field. Through the application of this architecture, based on active hardware redundancy, a gain in reliability, mean-lifetime and production yield is possible, while functionality is not degraded. System modeling in this thesis enables efficiency calculations for the presented architecture considering the additional hardware for redundancy and their administrative components. Therefore an efficient selection process for processor components and their amount of redundancy is possible. Consequently, the optimal amount of redundancy for a preexisting system and an objective to achieve can be calculated and is furthermore available early in the design process.

Towards describing structure as well as functionality of the architecture this thesis show that the integration in existing design processes with usual methods and tools is possible. The used system modeling, which realizes the redundancy selection process, is described as well. Finally, an application example is used to exhibit the practicability of the presented approach. The resulting efficiency and the required costs of this approach for the chosen example are discussed, too.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Die Entwicklung der Komplexität integrierter Schaltungen (ICs) wurde bereits im Jahre 1965 durch Gordon E. Moore abgeschätzt [Moo98]. Die von ihm aufgestellten Moorschen Gesetze sind Schlussfolgerungen aus damaligen Beobachtungen und sagen unter anderem eine Verdopplung der Komplexität von integrierten Schaltungen alle 18 bis 24 Monate voraus. Die *International Technology Roadmap for Semiconductors* (ITRS), ein Konsortium weltweit führender Halbleiterhersteller, welches Prognosen zur zukünftigen Entwicklung in der Halbleiterindustrie abgibt, sagt diese Entwicklung auch für die nächsten Jahre voraus [ITR10, ITR11]. Die stetige Skalierung der Fertigungstechnologien führt zu einer höheren Integrationsdichte und bei gleichbleibender Fläche der ICs zu einer höheren Anzahl von Transistoren pro Chip. Durch diesen Fortschritt eröffnen sich immer neue Möglichkeiten der Integration komplexerer Funktionen in Hardware. Diese Entwicklung in der Halbleiterelektronik führt allerdings auch dazu, dass die Anfälligkeit der Schaltungen gegenüber transienten, intermittierenden und permanenten Fehlern, bedingt durch physikalische Effekte während der Produktion und im Feld, zunimmt [ABMC08]. Hierdurch steigt auch die Notwendigkeit der Verwendung von Fehlertoleranz zur Kompensation dieser Effekte [ITR11]. Neben Verfahren für transiente Fehler wird die Behandlung von permanenten Fehlern in ICs zunehmend wichtig.

1.1 Motivation

Der Einsatz von Fehlertoleranz zur Kompensation von permanenten Fehlern in hochintegrierten Baugruppen wie Speichern und Caches ist längst Stand der Technik [BSOS04, KZK+98, PASM08, TWH+07, HCW06, LYHW03]. Gründe für diesen Einsatz sind zum einen die hochgradige Regularität, welche zur hohen Effizienz

beiträgt und damit auch eine Kompensation hoher Fehlerraten erlaubt [NAA04], und zum anderen die höhere Anfälligkeit von Speichern gegenüber Fehlereffekten aufgrund ihrer oft geringeren Strukturgrößen und der damit verbundenen höheren Integrationsdichte [ITR11]. Verfahren zur Kompensation permanenter Fehler in Speichern tragen damit ganz wesentlich zu einer höheren Ausbeute der Produktion und einer höheren Zuverlässigkeit von SoCs (*System-on-a-Chip*) bei. Hingegen sind Maßnahmen der Fehlertoleranz zur Steigerung der Ausbeute und Erhöhung der Zuverlässigkeit bei Prozessoren noch nicht auf dem Stand von Techniken für Speicher. Damit zukünftige integrierte Schaltungen weiter profitabel produziert und in zuverlässigen und langlebigen Systemen eingesetzt werden können, bedarf es zusätzlich zu den bereits existierenden und zum Teil sehr effizienten Maßnahmen der Fehlertoleranz für Speicher auch einer erhöhten Anstrengung bei der Entwicklung von Möglichkeiten der Kompensation permanenter Fehler in Prozessoren.

Die Notwendigkeit rekonfigurierbarer Funktionen, unter anderem auch zur Kompensation von permanenten Fehlern, wird ebenfalls durch die ITRS vorausgesagt [ITR07]. In Abbildung 1.1 ist der Anteil rekonfigurierbarer Funktionen für ein SoC bis zum Jahr 2023 dargestellt. Demnach soll der Anteil rekonfigurierbarer Funktionen in SoCs von heutigen 38% bis auf 69% im Jahr 2023 steigen. Dabei sind herstellbare Lösungen, welche einen Anteil von 56% der Funktionen in SoC durch Rekonfiguration erreichen, bis heute nicht bekannt.

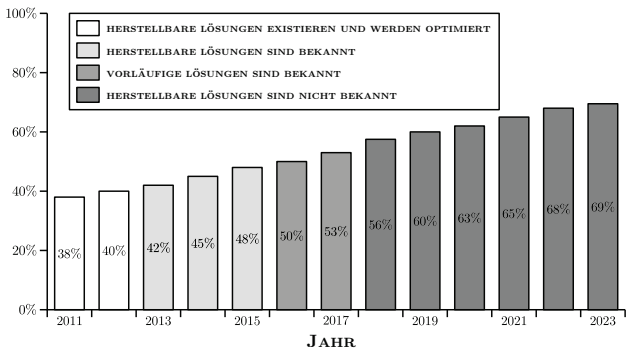


Abbildung 1.1: Prozentualer Anteil rekonfigurierbarer Funktionen in SoCs (vgl. [ITR07])

Gerade die fehlende Regularität in Prozessoren, vor allem für CISC[1] basierte Befehlssätze, schränkt die Möglichkeiten der Kompensation permanenter Fehler

[1]Complex Instruction Set Computer - bezeichnet einen Rechner mit komplexen Befehlssatz

durch Hardware-Redundanz stark ein [DQ11]. Superskalare Prozessoren mit einem reduzierten Befehlssatz (RISC[2]) bieten aufgrund ihrer einfacheren Struktur und höheren Regularität bezüglich einer effizienten Verwendung von Hardware-Redundanz mehr Potential als Prozessoren mit einem komplexen Befehlssatz.

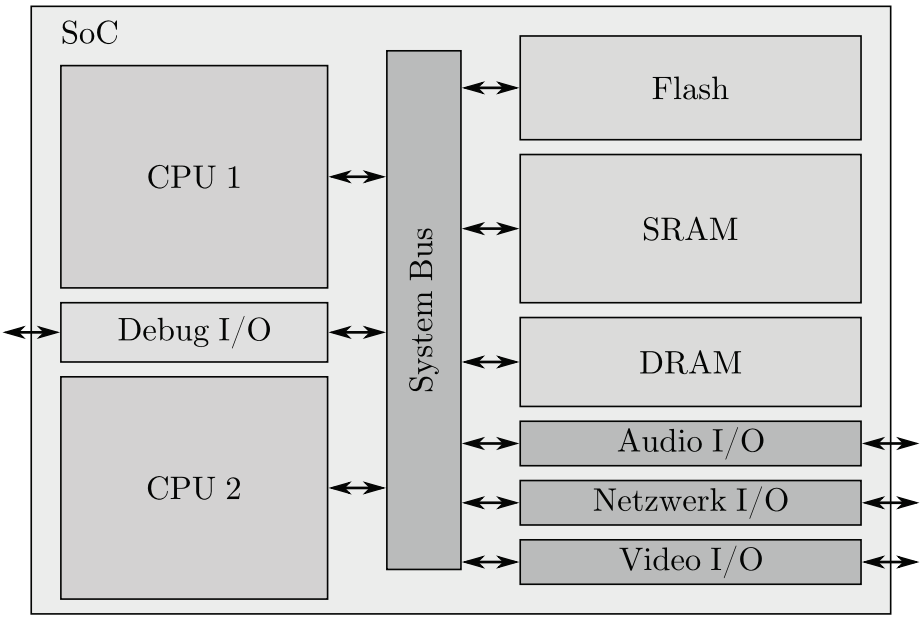


Abbildung 1.2: Blockdiagramm eines System-on-a-Chip

In Abbildung 1.2 ist ein Blockdiagramm eines SoC dargestellt. Alle oder zumindest der größte Teil der notwendigen Funktionen sind typischerweise auf einem Chip integriert. Neben den Prozessoren und einer ganzen Reihe an Ein-/Ausgabe Schnittstellen beinhaltet ein SoC oft auch unterschiedliche Arten von Speichern. Nachdem der ursprüngliche Anteil von Speichern in SoC Lösungen zum Ende des 20. Jahrhunderts eher gering war, hat er sich seitdem zu einem wesentlichen Anteil der Gesamtfläche entwickelt [MPKSZ05]. Für das Gesamtsystem sind Maßnahmen der Fehlertoleranz für Speicher damit zwingend notwendig, um mit der steigenden Anfälligkeit von integrierten Schaltungen umzugehen. Allerdings ist deren Effizienz, gerade für Verfahren die eine Kompensation für hohe Fehlerraten erlauben, kein Garant für hohe Produktionsausbeuten oder lange Einsatzzeiten, solange keine geeigneten Verfahren speziell für Prozessoren existieren, die ähnliche Eigenschaften besitzen.

[2]Reduced Instruction Set Computer - bezeichnet einen Rechner mit reduziertem Befehlssatz

1.2 Zielsetzung der Arbeit

Diese Arbeit stellt ein Verfahren zur Kompensation permanenter Fehler durch aktive Hardware-Redundanz für beliebige Komponenten von Prozessoren vor, welches direkt nach der Produktion oder auch im Feld zum Einsatz kommen kann. Somit soll dem beschriebenen Ungleichgewicht bei der Kompensation von permanenten Fehlern zwischen Speichern und Prozessoren begegnet werden und im Ergebnis zu einer besseren Fehlerkompensation für SoC Lösungen beigetragen werden. Eine effiziente Auswahl von Komponenten, einschließlich der Bewertung unterschiedlicher Redundanzen, ist vor allem durch die gegebene geringere Regularität von Prozessoren notwendig, um eine optimierte Lösung für ein spezielles Problem zu finden. Mögliche Probleme sind eine nicht ausreichende Zuverlässigkeit, zu geringe Lebensdauer oder schlechte Produktionsausbeuten. Für den Entwurf von Prozessoren mit Redundanz muss diese Bewertung so zeitig wie möglich im Entwurfsprozess erfolgen, und die Umsetzung muss sich ebenfalls in bestehende Entwurfsabläufe integrieren lassen.

Arbeiten zur effizienten Auswahl von Komponenten beispielsweise für VLIW Prozessoren liegen bereits vor und leisten eine Entwurfsraumexploration für verschiedene Signalverarbeitungsalgorithmen [Jun11, Sch06]. Solche optimierten Lösungen sind dann zwar in der Lage bestimmte Algorithmen effizient umzusetzen, jedoch werden mögliche Ausfälle von Baugruppen durch Produktions- oder Alterungsfehler nicht berücksichtigt. Dies kann dazu führen, dass notwendige Eigenschaften wie zu erreichende Zuverlässigkeit, Lebensdauer oder Produktionsausbeute nicht erfüllt werden und damit der Einsatz dieser Lösungen fraglich wird.

Ausgehend von vorhandenen optimierten Lösungen für Prozessoren soll diese Arbeit eine weitere Optimierung liefern, welche sich speziell mit der Erfüllung zusätzlicher Eigenschaften, wie

- Erreichen einer bestimmten Zuverlässigkeit,
- Maximierung der mittleren Lebensdauer oder
- Steigerung der Produktionsausbeute,

beschäftigt.

1.3 Organisation der Arbeit

Die vorliegende Arbeit unterteilt sich in sieben Kapitel. Im nächsten Kapitel wird der Begriff der Zuverlässigkeit, ihre Eigenschaften, Kenngrößen und mögliche Bedrohungen erläutert. Weiterhin werden Möglichkeiten der Systemmodellierung zur Ermittlung relevanter Kenngrößen vorgestellt und eine detaillierte Betrachtung von Fehlerursachen durchgeführt. Der Stand der Technik von Maßnahmen der Fehlertoleranz, insbesondere für permanente Fehler, und Möglichkeiten des *variation aware design* werden in Kapitel 3 vorgestellt. Das 4. Kapitel beschäftigt sich mit der spezifischen Umsetzung einer skalierbaren aktiven Hardware-Redundanz für Komponenten in Prozessoren und ihrer Administration. Gleichzeitig wird die Integration des Verfahrens in bestehende Entwurfsprozesse aufgezeigt. In Kapitel 5 wird die Systemmodelleriung dieser Arbeit zur Bewertung verschiedener Implementierungen der im 4. Kapitel vorgestellten skalierbaren Hardware-Redundanz beschrieben. Anschließend folgt im 6. Kapitel die Realisierung für ein Anwendungsbeispiel einschließlich der Ergebnisse des optimierten Auswahlprozesses zur Steigerung von Zuverlässigkeit, mittlerer Lebensdauer oder Produktionsausbeute. Abgeschlossen wird die vorgelegte Arbeit mit einer Zusammenfassung und einem Ausblick im letzten Kapitel.

KAPITEL 2

Grundlagen

Dieses Kapitel beinhaltet eine Begriffsklärung und die Erläuterung notwendiger Grundlagen für diese Arbeit. Die Begriffsklärung umfasst im Wesentlichen Erläuterungen zu den Bedeutungen der Zuverlässigkeit und ihren einzelnen Eigenschaften. Anschließend werden die für diese Arbeit relevanten Kenngrößen der Zuverlässigkeit vorgestellt. Danach werden mögliche System-Modellierungen beschrieben, welche wesentlicher Bestandteil der Ermittlung der jeweiligen Kenngrößen sind. Im letzten Teil dieses Kapitels erfolgt eine detaillierte Darstellung möglicher Fehlerursachen integrierter Schaltungen.

2.1 Zuverlässigkeit

Der Begriff der Zuverlässigkeit wird in der Fachsprache mit unterschiedlicher Bedeutung verwendet. Im allgemeinen wird die Zuverlässigkeit (*dependability*) als Sammelbegriff für verschiedene Eigenschaften eines technischen Systems verwendet. Die Bedeutungen der Zuverlässigkeit sind, abhängig vom jeweiligen System, sehr unterschiedlich. In Tabelle 2.1 (vgl. [Kon07]) sind mögliche Bedeutungen der Zuverlässigkeit beschrieben.

Lebenswichtige Anlagen	Anlagen deren Ausfall Gesundheit und Leben gefährden können. Beispielhaft seien hier Verkehrssteuerungen oder medizinische Anlagen erwähnt.

Sehr teure Anlagen	Ein Ausfall solcher Anlagen kann dazu führen, dass jahrelange Arbeit zunichte gemacht wird. Hier sind Anlagen der Raumfahrt, Satelliten oder auch Forschungsapparaturen gemeint, deren Ausfall finanziell nicht zu verkraften ist.
Sehr komplexe Anlagen	Komplexere Systeme mit vielen Baugruppen erreichen weder lange Einsatzzeiten noch eine hohe Zuverlässigkeit ohne Maßnahmen zur Lebenszeit- oder Zuverlässigkeitssteigerung im Vorfeld zu ergreifen.
Weniger Reparaturen	Zuverlässigere Systeme haben, durch weniger anfallende Reparaturen, weniger Reparaturkosten und damit geringere Stillstandzeiten.
Billigere Anlagen	Die zum Teil hohen Anschaffungskosten von Anlagen, welche Maßnahmen der Zuverlässigkeitssteigerung beinhalten, kann sich durch geringere und weniger Reparaturen im Einsatz amortisieren.
Sichere Anlagen	Zuverlässigere Anlagen bedeuten nicht immer eine Lebenszeitsteigerung, aber sie bedeuten immer höhere Sicherheit.

Tabelle 2.1: Bedeutungen der Zuverlässigkeit (vgl. [Kon07])

Daraus wird ersichtlich, dass der Einsatz zuverlässiger Systeme in vielen Bereichen notwendig ist oder zu einer effizienteren Nutzung vorhandener Ressourcen beiträgt. Unter Zuverlässigkeit versteht man unter anderem verschiedene Eigen-

schaften eines technischen Systems. Dabei ist es vom jeweiligen Einsatz des Systems abhängig, welche Eigenschaften von besonders hoher Bedeutung sind. Die Autoren in [ALR04] definieren die folgenden Eigenschaften für die Zuverlässigkeit, welche in Tabelle 2.2 aufgelistet und erläutert sind.

Verfügbarkeit (*availability*)	Das System ist bereit seine Aufgabe korrekt zu erledigen.
Funktionszuverlässigkeit (*reliability*)	Das System liefert kontinuierlich korrekte Ergebnisse.
Sicherheit (*safety*)	Katastrophale Folgen für Nutzer und Umgebung bleiben aus. (berücksichtigt insbesondere missbräuchliche Nutzung)
Integrität (*integrity*)	Keine ungewünschten Systemveränderungen treten auf.
Wartbarkeit (*maintainability*)	Möglichkeiten von Modifikationen bzw. Reparaturen sind vorhanden.
Vertraulichkeit (*confidentiality*)	Keine unautorisierten Verluste von Informationen.

Tabelle 2.2: Eigenschaften der Zuverlässigkeit (vgl. [ALR04])

Einzelne Eigenschaften der Zuverlässigkeit aus Tabelle 2.2 können mit Hilfe einer Zuverlässigkeitsanalyse bestimmt werden. Diese Zuverlässigkeitsanalyse umfasst dann eine Modellierung des Systems zur Bestimmung der relevanten Kenngröße einer bestimmten Eigenschaft.

Zusätzlich zu den Eigenschaften der Zuverlässigkeit gibt es Bedrohungen, welche eine Verringerung der Zuverlässigkeit zur Folge haben können. Im Detail kann eine Bedrohung eine oder mehrere Eigenschaften beeinflussen. Mögliche Bedrohungen und ihre Erläuterungen sind in Tabelle 2.3 aufgelistet.

Fehler (*fault*)	Ein Fehler beschreibt allgemein die Ursache einer Störung.
Störung (*error*)	Beschreibt eine Abweichung des internen Systemzustandes, welche sich nicht zwingend zu einem Ausfall des Systems ausweitet.
Ausfall (*failure*)	Beschreibt eine Störung die zu einem Ausfall des System führt.

Tabelle 2.3: Bedrohungen der Zuverlässigkeit (vgl. [ALR04])

Um bestimmte Eigenschaften der Zuverlässigkeit zu erfüllen, müssen Maßnahmen ergriffen werden, welche in der Lage sind, die eben vorgestellten Bedrohungen aus Tabelle 2.3 zu verringern oder sogar zu beseitigen. Mögliche Maßnahmen zur Erhöhung der Zuverlässigkeit sind in Tabelle 2.4 aufgelistet und beschrieben.

Prevention (*prevention*)	Bezeichnet Maßnahmen, die das Auftreten von Fehlern verhindern.
Fehlerbehandlung (*fault removal*)	Bezeichnet Maßnahmen zur Reduzierung des Ausmaßes von Fehlern.
Vorhersage (*forecasting*)	Bezeichnet Maßnahmen, die eine Abschätzung über die Wahrscheinlichkeit von Anzahl, Vorkommen und möglichen Konsequenzen von Fehlern erlauben.
Toleranz (*tolerance*)	Bezeichnet Maßnahmen, die trotz Fehlern einen Ausfall verhindern.

Tabelle 2.4: Maßnahmen zur Steigerung der Zuverlässigkeit (vgl. [ALR04])

Zusammenfassend ist in Abbildung 2.1 eine Klassifizierung des Begriffes der Zuverlässigkeit, einschließlich ihrer Eigenschaften, Bedrohung und Maßnahmen, dargestellt.

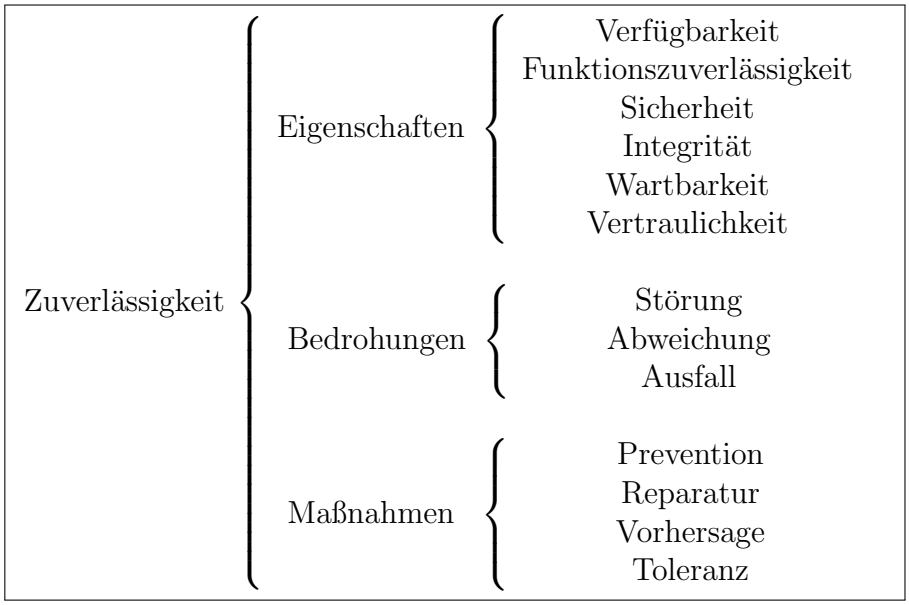


Abbildung 2.1: Taxonomie der Zuverlässigkeit angelehnt an [ALR04]

Zuverlässigkeitsanalysen sind aus heutiger Sicht für Hersteller von integrierten Schaltungen nicht mehr wegzudenken. Die Gründe dafür sind zum einen die steigende Komplexität sowohl des Herstellungsprozesses als auch des Funktionsumfangs der ICs. Gleichzeitig steigt die Anfälligkeit gegenüber verschiedenen Fehlereffekten durch die stetige Skalierung der Schaltungsparameter zu kleineren Dimensionen. Die Zuverlässigkeitsanalysen dienen dazu, bestimmte Eigenschaften eines Systems zu bewerten, um anschließend den Einsatz verschiedener Maßnahmen und ihre Effizienz beurteilen zu können.

In dieser Arbeit wird die Maßnahme der Fehlertoleranz angewendet, um auf die Bedrohung der steigenden Anfälligkeit von Schaltungen gegenüber Fehlern zu reagieren. Der Begriff Fehler bezieht sich hier explizit auf solche Fehler die eine dauerhafte Störung verursachen. Die Bewertung der verwendeten Fehlertoleranz wird durch einen Vergleich bestimmter Kenngrößen durchgeführt. Die Bestimmung der jeweiligen Kenngrößen erfordert eine Modellierung der Funktionszuverlässigkeit.

Im Folgenden werden zuerst die verwendeten Kenngrößen beschrieben. Anschließend wird auf verschiedene Möglichkeiten eingegangen, welche Modellierungen der Eigenschaft der Funktionszuverlässigkeit bereitstellen und damit die Ermittlung der Kenngrößen erlauben.

2.1.1 Funktionszuverlässigkeit

Eine Eigenschaft der Zuverlässigkeit (*dependability*) kann durch das Maß der Funktionszuverlässigkeit (*reliability*) sowohl qualitativ als auch quantitativ beschrieben werden. Im folgenden wird ausschließlich der Begriff der Zuverlässigkeit verwendet und ist in dieser Arbeit gleichbedeutend mit der Funktionszuverlässigkeit.

Die Zuverlässigkeit (*reliability*) $R(t)$ ist definiert als die Wahrscheinlichkeit, dass ein System im Interval $[0, t]$ seine Funktion erfüllt.

Da es sich bei der Zuverlässigkeit $R(t)$ um eine Wahrscheinlichkeit handelt, müssen stets alle Axiome von Kolmogorow erfüllt sein.

1. Die Wahrscheinlichkeit $P(A)$ eines Ereignisses A ist stets eine reelle Zahl zwischen 0 und 1 : $0 \leq P(A) \leq 1$
2. Die Wahrscheinlichkeit des sicheren Ereignisses ist 1 : $P(\Omega) = 1$
3. Die Wahrscheinlichkeit der Vereinigung zweier unvereinbarer Ereignisse A und B ist gleich der Summe der Einzelwahrscheinlichkeiten:
   $A \cap B = \emptyset \Rightarrow P(A \cup B) = P(A) + P(B)$

Die Zuverlässigkeit oder auch Überlebenswahrscheinlichkeit $R(t)$ wird durch eine kumulative Verteilungsfunktion oder auch Unzuverlässigkeit $F(t)$ beschrieben. Im Folgenden ist der Zusammenhang zwischen $R(t)$ und $F(t)$ dargestellt.

$$R(t) = 1 - F(t) \tag{2.1}$$

Zur Beschreibung der stetigen Verteilungsfunktion $F(t)$ wird eine Wahrscheinlichkeitsdichtefunktion $f(t)$ benötigt. Diese beschreibt die Änderungen der Verteilungsfunktion $F(t)$ in einem infinitesimalen Intervall und ist damit folgendermaßen definiert.

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt} \tag{2.2}$$

Die Funktion der Fehlerrate $h(t)$ bezeichnet das Verhältnis von Wahrscheinlichkeitsdichte und Zuverlässigkeit.

$$h(t) = \frac{f(t)}{1 - F(t)} = \frac{f(t)}{R(t)} \tag{2.3}$$

Im Bereich der Halbleiterelektronik hat sich der Begriff FIT (*Failure in Time*) als Einheit für die Fehlerrate etabliert. Dabei entspricht 1 FIT genau einem Fehler in 10^9 Stunden Einsatzzeit.

2.1.2 Mittlere Lebensdauer

Die mittlere Lebensdauer ($MTTF$[1]) bezeichnet eine weitere statistische Kenngröße, mit der die mittlere Zeit bis zum Ausfall eines Systems beschrieben werden kann.

$$MTTF = \int_0^\infty R(t)\mathrm{d}t = \int_0^\infty tf(t)\mathrm{d}t \tag{2.4}$$

Die Verwendung der Kenngröße $MTTF$ geht von einem System aus, welches keine Möglichkeiten der Reparatur zulässt. Für Systeme, bei denen eine Reparatur möglich beziehungsweise angedacht ist, wird häufig der Begriff der mittleren Zeit zwischen Fehlern ($MTBF$[2]) verwendet. Diese Größe beschreibt die durchschnittliche Zeit zwischen zwei Ausfällen und ist in Gleichung 2.5 definiert.

$$MTBF = \frac{\text{Gesamtbetriebszeit}}{\text{Anzahl aller Ausfälle}} \tag{2.5}$$

Mit Hilfe der mittleren Zeit bis zur Reparatur ($MTTR$[3]) wird die durchschnittliche Reparaturzeit beschrieben. Diese ist in Gleichung 2.6 definiert.

$$MTTR = \frac{\text{Gesamtreparaturzeit}}{\text{Anzahl aller Reparaturen}} \tag{2.6}$$

Falls die Reparatur ohne Verzögerung nach dem Ausfall beginnt, dann kann der Zusammenhang zwischen $MTTF$, $MTBF$ und $MTTR$ folgendermaßen dargestellt werden:

$$MTBF = MTTF + MTTR \tag{2.7}$$

Im Verlauf dieser Arbeit wird ausschließlich die Kenngröße $MTTF$ Verwendung finden. Während der System-Rekonfiguration in dieser Arbeit gilt das System als funktionstüchtig und nicht Ausgefallen. Ein Ausfall des Systems in dieser Arbeit ist der Zeitpunkt an dem eine Rekonfiguration keinen Zustand herstellen kann, für den dauerhafte Störungen als kompensiert gelten.

[1] Mean Time To Failure
[2] Mean Time Between Failures
[3] Mean Time To Recover

2.1.3 Verteilungsfunktionen

Verteilungsfunktionen dienen in der Wahrscheinlichkeitstheorie dazu den Wert einer Zufallsvariablen zu beschreiben. In technischen Systemen versucht man diese Verteilungsfunktion (bzw. Ausfallverteilung) mit Hilfe empirischer Daten über Ausfälle oder Störungen zu ermitteln. Diese Herangehensweise ist nicht immer praktikabel, etwa für Anlagen, die sehr wenige bis keine Ausfälle aufweisen, wie beispielsweise Atomkraftwerke oder Satelliten, oder für Anlagen, die noch nicht in Betrieb genommen wurden.

Eine weitere Besonderheit für viele technische Systeme besteht darin, dass die Ausfallverteilung über die Lebenszeit des Systems variiert. Dieser Effekt wird durch den Badewanneneffekt bzw. die Badewannenkurve beschrieben und ist in Abbildung 2.2 dargestellt. Auf der Y-Achse ist die Fehlerrate und auf der X-Achse die Zeit abgebildet.

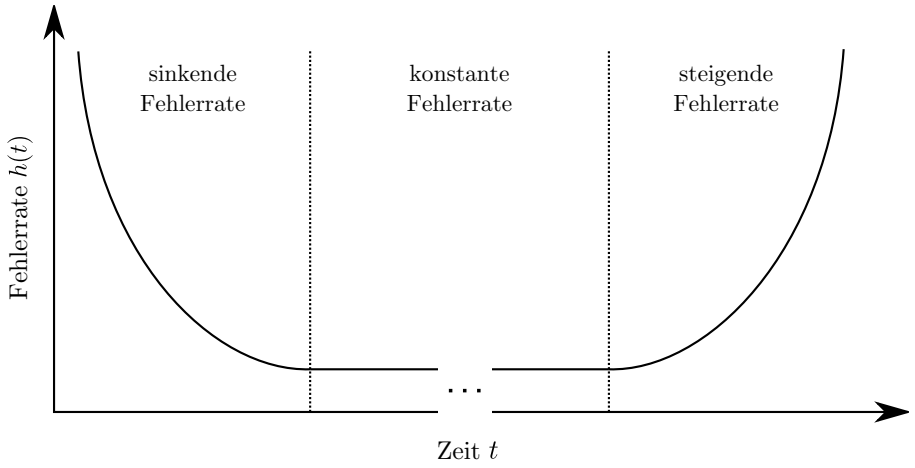


Abbildung 2.2: Badewannenkurve für technische Systeme

Grundsätzlich besteht die Badewannenkurve aus 3 Phasen die sich durch unterschiedliche Fehlerraten und damit durch verschiedenen Ausfallverteilungen beschreiben lassen.

1. In der ersten Phase sinkt die Fehlerrate (zeitige Ausfälle).
2. In der zweiten Phase bleibt die Fehlerrate konstant (zufällige Ausfälle).
3. In der letzten Phase steigt die Fehlerrate (altersbedingte Ausfälle).

Die Unterteilung in drei Phasen gilt für viele technische Systeme ist aber auch auf andere Systeme wie beispielsweise den Menschen anwendbar. Im Folgenden wird die Weibull Verteilung und einer ihrer Spezialfälle, die Exponentialverteilung, vorgestellt. Alle drei Phasen der Badewannenkurve lassen sich mit dieser Verteilungsfunktion modellieren.

Weibull Verteilung

Die Weibull Verteilung, benannt nach dem Schweden Waloddi Weibull, wird häufig zur Modellierung verschiedener Lebensphasen eines Produktes ausgewählt. Durch eine gezielte Auswahl ihrer Parameter lassen sich Systeme mit sinkender, steigender und auch konstanter Fehlerrate abbilden. Die Dichte der Funktion ist gegeben durch:

$$f(t) = \alpha \cdot \beta \cdot (\alpha \cdot t)^{\beta-1} \cdot e^{-(\alpha \cdot t)^{\beta}}, \tag{2.8}$$

ihre Verteilungsfunktion ist damit:

$$F(t) = 1 - e^{-\alpha \cdot t^{\beta}}, \tag{2.9}$$

und ihre Fehlerrate lautet:

$$h(t) = \alpha \cdot \beta \cdot t^{\beta-1}. \tag{2.10}$$

Hier beschreibt α einen Skalierungsfaktor und β den Formfaktor. Die Auswahl des Formfaktors β bestimmt maßgeblich die Entwicklung der Ausfallverteilung über die Zeit. Folgende Parameter werden zur Modellierung der verschiedenen Phasen der Badewannenkurve verwendet:

- $0 < \beta < 1$ beschreibt die Frühausfälle, die mit fortlaufender Zeit abnehmen,
- $\beta = 1$ beschreibt Zufallsausfälle, die konstant über die Zeit sind (Exponentialverteilung),
- $\beta > 1$ beschreibt die Ermüdungs- bzw. Verschleißausfälle, die mit fortlaufender Zeit zunehmen.

In Abbildung 2.3 sind die resultierenden Fehlerraten (Y-Achse) in Abhängigkeit der Zeit (X-Achse) für unterschiedliche Werte des Formfaktors β der Weibull Verteilung dargestellt. Abbildung 2.4 zeigt anschließend die resultierende Wahrscheinlichkeitsdichte (Y-Achse) in Abhängigkeit der Zeit (X-Achse) ebenfalls für

verschiedene Formfaktoren β der Weibull Verteilung. Der Formfaktor wurde so gewählt, dass die Auswirkung von fallenden, konstanten und auch steigende Fehlerraten auf die Wahrscheinlichkeitsdichte sichtbar werden.

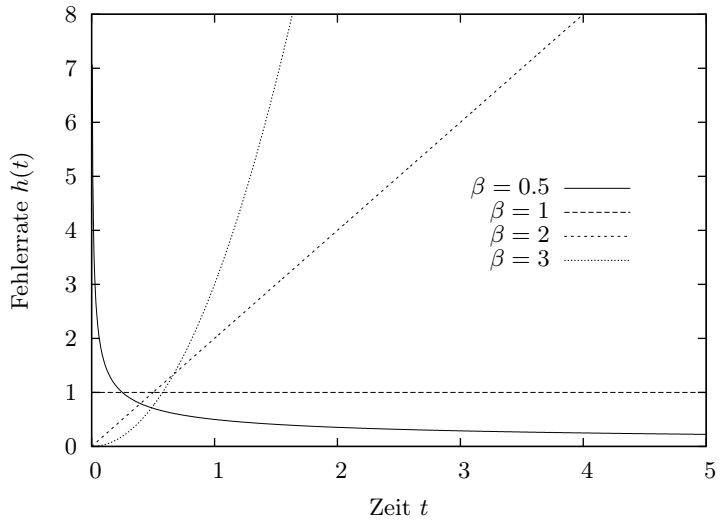


Abbildung 2.3: Fehlerrate der Weibull Verteilung

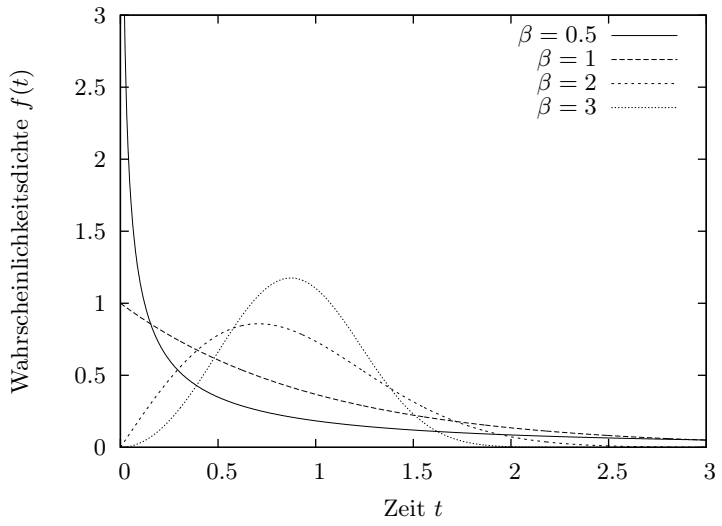
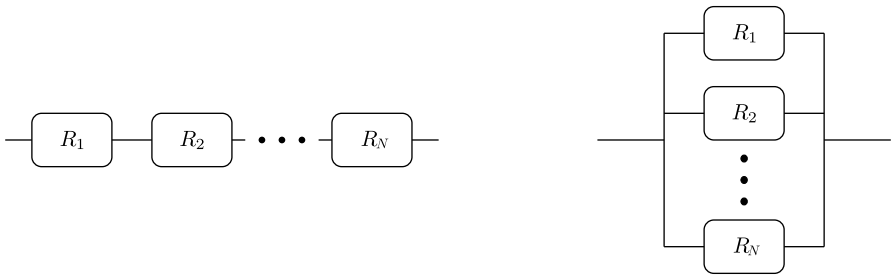


Abbildung 2.4: Wahrscheinlichkeitsdichte der Weibull Verteilung

2.2 Möglichkeiten der Systemmodellierung

2.2.1 Modellierung der Zuverlässigkeit

Die Zuverlässigkeit eines Systems kann durch unterschiedliche Modelle beschrieben werden. In dieser Arbeit wird die Zuverlässigkeit mit dem *reliability block diagram* (RBD) modelliert [RH03]. Durch die Modellierung mit RBD können technische Systeme als logische Strukturen, in Bezug auf die Erfüllung ihrer Funktion, beschrieben werden.



(a) Serien System (b) Parallel System

Abbildung 2.5: RBD für ein Serien- und Parallelsysteme

Ein komplexes System wird durch die Zuverlässigkeit seiner Subsysteme als Komposition aus Serien- und Parallelsystemen dargestellt. Im einfachsten Fall müssen alle Komponenten intakt sein, damit das Gesamtsystem funktioniert. Diese Eigenschaft kann durch ein Seriensystem (dargestellt in Abbildung 2.5(a)) beschrieben werden. Die Zuverlässigkeit eines solchen Systems entspricht dann dem Produkt der einzelnen Zuverlässigkeiten der Komponenten (Gleichung 2.11).

$$R_{System}(t) = \prod_{i=1}^{N} R_i(t) \tag{2.11}$$

Besteht das Gesamtsystem aus mehreren unabhängigen Komponenten, von denen eine intakte Komponente die Funktion des Gesamtsystems noch gewährleistet, dann kann es durch ein Parallelsystem (dargestellt in Abbildung 2.5(b)) beschrieben werden. Die Zuverlässigkeit eines Parallelsystems wird durch Gleichung 2.12 bestimmt.

$$R_{System}(t) = 1 - \prod_{i=1}^{N}(1 - R_i(t)) \tag{2.12}$$

In der Realität lassen sich jedoch nicht alle Systeme durch Serien- bzw. Parallelsysteme beschreiben. Die Autoren in [KK07] beschreiben für diese Strukturen ein Verfahren basierend auf Erweiterungen des Modells durch Fallunterscheidungen.

Eine weitere Möglichkeit die Zuverlässigkeit dieser Systeme zu bestimmen, ist die Verwendung einer oberen bzw. unteren Schranke für die Zuverlässigkeit. Eine obere Schranke für die Zuverlässigkeit ist gegeben durch:

$$R_{System}(t) \leq 1 - \prod_{i=1}^{P}(1 - R_{P_i}(t)) \tag{2.13}$$

In Gleichung 2.13 beschreibt R_{P_i} die Zuverlässigkeit eines Serienpfades i durch das Blockdiagramm. Hierfür wird angenommen, dass alle Pfade voneinander unabhängig sind. Aus diesem Grund gibt die Gleichung nur die obere Schranke für die Zuverlässigkeit an, da in der Praxis oft mehrere Pfade identische Komponenten benutzen. Eine untere Schranke ist ebenfalls definiert durch:

$$R_{System}(t) \geq \prod_{i=1}^{C} R_{C_i}(t) \tag{2.14}$$

Die untere Schranke beschreibt die Zuverlässigkeit des Systems anhand einer minimalen Menge von Komponenten die ausfallen müssen, damit das System nicht mehr funktioniert. Hier bezeichnet R_{C_i} die Zuverlässigkeit einer Menge von Komponenten deren Ausfall einen Systemausfall bedeuten. Auch hier wird angenommen, dass alle minimalen Mengen voneinander unabhängig sind. Diese Annahme führt genau dann zur unteren Schranke, wenn mindestens eine Komponente in unterschiedlichen Mengen vorkommt.

Ein *M aus N* System besteht aus N Komponenten, von denen mindestens M intakt sein müssen, damit das Gesamtsystem funktionsfähig bleibt. Die Zuverlässigkeit für ein beliebiges *M aus N* System ist in Gleichung 2.15 definiert.

$$R_{M\ aus\ N}(t) = \sum_{i=M}^{N} R^i(t)[1 - R(t)]^{N-i} \tag{2.15}$$

Die Modellierung eines Systems mit RBD stellt eine einfach anzuwendende Möglichkeit dar, die Zuverlässigkeit des Gesamtsystems aus der Zuverlässigkeit seiner Subsysteme zu ermitteln.

Weitere Modellierungen der Zuverlässigkeit

Die vorgestellten *reliability block diagrams* sind nur eine von vielen Möglichkeiten der Zuverlässigkeitsmodellierung. In diesem Abschnitt sollen kurz weitere vorgestellt werden. Eine Erweiterung der RBD sind die in [DX06] vorgestellten DRBD (*dynamic reliability block diagram*). DRBD sind in der Lage, verschiedene Zustände der einzelnen Komponenten und ihre Abhängigkeiten zu anderen Komponenten zu modellieren.

Eine weitere Alternative zur Modellierung stellen die sogenannten FT (*fault-trees*) dar. Diese Art der Modellierung beschreibt, im Gegensatz zu den RBD, wann ein System nicht mehr funktioniert. Hier werden Komponenten als Knoten dargestellt. Serienverbindungen von Komponenten werden mit OR Gattern verbunden und parallele Strukturen mit AND Gattern. Jeder defekte Knoten produziert eine logische 1 am jeweiligen Gattereingang. Sobald das oberste Gatter den Wert 1 besitzt, ist das Gesamtsystem nicht mehr funktionstüchtig. Auch dieses Modell erfuhr zahlreiche Erweiterungen. So existieren unterschiedliche Gatter wie NOT, XOR oder auch priorisierte AND Gatter, um die Systemeigenschaften genauer modellieren zu können. Die Autoren in [MDCS98] erweiterten die *fault-trees* mit Hilfe von BDD (*binary decision diagrams*) und Markov Modellen zu DFT (*dynamic fault-trees*).

Die Modellierung der Zuverlässigkeit mit Markov Ketten stellt eine weitere Möglichkeit dar. Markov Ketten basieren auf einem Zustandsraum. Alle Zustände repräsentieren unterschiedliche Gegebenheiten des Systems, zum Beispiel bestimmte Fehlersituationen. Die Zustandsübergänge beschreiben bestimmte Ereignisse, welche im System ablaufen können. In Markov Ketten sind die Zustandsübergänge nur vom aktuellen Zustand abhängig und nicht von den Zuständen davor.

2.2.2 Modellierung der mittleren Lebensdauer

Die Ermittlung der $MTTF$ kann durch Gleichung 2.4 erfolgen. Dies setzt voraus, dass die Zuverlässigkeit $R(t)$ des Systems bereits modelliert wurde. In der Praxis erfolgt die Ermittlung der $MTTF$ oft empirisch.

In der Literatur finden sich einige weitere Möglichkeiten, welche sich mit der Modellierung der $MTTF$ beschäftigen [SABR04a, Sri06, SZH+07, Shi08, HX10a]. Alle diese Herangehensweisen sollen während des Entwurfsprozesses von ICs den Entwicklern eine zeitige Möglichkeit bereitstellen, unterschiedliche Modellierun-

gen eines Systems in Bezug auf die mittlere Lebensdauer zu bewerten. Mit RAMP (*reliability aware micro processor*) [SABR04a, Sri06] wird ein Modell vorgestellt, welches unter Berücksichtigung verschiedener Fehlereffekte die Ermittlung der $MTTF$ erlaubt. Dazu wird eine Summe der Fehlerraten ermittelt. Somit können unterschiedliche Implementierungen eines Systems und die Auswirkungen auf die $MTTF$ berechnet werden. Die verwendete Methode der Summe von Fehlerraten gilt nur unter folgenden Voraussetzungen:

- Jeder in Betracht gezogene Fehlereffekt besitzt eine konstante Fehlerrate.
- Die Fehlerraten aller Fehlereffekte werden als gleichverteilt angenommen.
- Das zu evaluierende Modell ist als Seriensystem beschrieben.

Die Autoren in [SZH+07] bezeichnen die ersten beiden Punkte der Auflistung als unbegründet, da durch die Verwendung von konstanten Fehlerraten eine Bewertung in Bezug auf Alterungseffekte nicht möglich ist und verschiedene Fehlereffekte durchaus nicht gleichverteilt sind. Viel entscheidender für die vorliegende Arbeit ist allerdings die Tatsache, dass Schaltungen mit Redundanzen durch die Modellierung als Seriensystem nicht bewertet werden können.

Eine Erweiterung des eben vorgestellten Modells hebt die Einschränkungen von konstanten Fehlerraten und Gleichverteilungen dieser für verschiedene Fehlereffekte auf [SZH+07, Shi08]. Dazu wird mit Hilfe einer einmaligen Charakterisierung von unterschiedlichen Referenzschaltungen der Einfluss verschiedener Fehlereffekte bestimmt. Damit entsteht eine Fehlerrate, die durch die jeweilige Referenzschaltung bestimmt wurde. Allerdings stößt auch diese Modellierung der mittleren Lebensdauer in Bezug auf Systeme mit Redundanz an ihre Grenzen [SZH+07].

Die Autoren in [HX10a] beschreiben ein simulationsbasiertes Verfahren (AgeSim) zur Ermittlung der mittleren Lebensdauer. Dabei wird versucht, die Nachteile von konstanten Fehlerraten der Modelle in [SABR04a] und [SZH+07] durch eine aussagekräftige Simulation mit anschließender Parameter-Extraktion zu beheben, um damit Aussagen über die resultierende $MTTF$ auch in Bezug auf Alterungseffekte für unterschiedliche Implementierungen treffen zu können. Auch eine theoretische Erweiterung ihres Ansatzes auf Schaltungen mit Redundanz wird vorgestellt. Somit meinen die Autoren eine gesteigerte Genauigkeit gegenüber den Modellen aus [SABR04a] und [SZH+07] zu erreichen. Jedoch ist dieses Verfahren äußerst aufwendig und benötigt eine aussagekräftige Simulation, welche entsprechend realistische Stimuli voraussetzt. Diese muss die typische Arbeitslast einer Schaltung über ihren gesamten Lebenszyklus abbilden.

2.2.3 Modellierung der Ausbeute

Die Ausbeute[4] bei der Produktion von ICs gibt an, wie viele Chips von allen produzierten Einheiten der geforderten Spezifikation entsprechen. Damit entspricht die Ausbeute dem Verhältnis in Gleichung 2.16.

$$\text{Ausbeute} = \frac{\text{Chips ohne (Produktions-)Fehler}}{\text{alle produzierten Chips}} \tag{2.16}$$

Jeder Hersteller ist bemüht diese Ausbeute so hoch wie möglich ausfallen zu lassen. Dazu wird systematisch versucht, Probleme im Herstellungsverfahren ausfindig zu machen und zu beseitigen. Diese Herangehensweise ist vor jeder Massenproduktion notwendig, um die Kosten der Herstellung zu minimieren. Allerdings lassen sich nicht alle Fehlerquellen durch eine Verbesserung des Herstellungsprozesses abschalten. Denn zusätzlich zu den systematischen Verlusten bei der Ausbeute sind es auch zufällige Effekte, welche zu Produktionsfehlern führen. Aus diesem Grund ist auch die Verwendung von Verfahren der Fehlertoleranz eine Möglichkeit, geringerer Ausbeute entgegenzuwirken [KS90]. Fehlertoleranz kann zu einer Steigerung der Ausbeute führen, wenn sich dadurch das Verhältnis aus Gleichung 2.16 verbessert. Dabei erweitert sich der Zähler aus Gleichung 2.16 um die Chips, welche durch die Maßnahme der Fehlertoleranz mögliche Produktionsfehler kompensieren.

Allerdings darf an dieser Stelle nicht vernachlässigt werden, dass zusätzliche Maßnahmen der Fehlertoleranz die Größe der einzelnen Chips ansteigen lässt. Somit reduziert sich die Anzahl aller Chips die auf einem Wafer[5] produziert werden können. Durch Gleichung 2.17 lässt sich die Anzahl aller Chips pro Wafer berechnen [HP07].

$$\text{Chips pro Wafer} = \frac{\pi \times (\text{Wafer-Durchmesser}/2)^2}{\text{Chip-Fläche}} - \frac{\pi \times \text{Wafer-Durchmesser}}{\sqrt{2 \times \text{Chip-Fläche}}} \tag{2.17}$$

Die Autoren in [HP07] beschreiben ein einfaches Modell zur Berechnung der Ausbeute eines IC. Dieses ist in Gleichung 2.18 beschrieben und setzt voraus, dass Defekte zufällig über den gesamten Wafer verteilt sind und die Ausbeute umgekehrt proportional zur Komplexität des Herstellungsprozesses ist [HP07].

[4] engl. yield

[5] Wafer - bezeichnen in der Halbleiterelektronik meist kreisrunde Scheiben die als Rohlinge zur Produktion integrierter Schaltungen verwendet werden

$$\text{Chip-Ausbeute} = \text{Wafer-Ausbeute} \times (1 + \frac{\text{Defekte pro Fläche} \times \text{Chip-Fläche}}{\alpha})^{-\alpha} \tag{2.18}$$

Hier beschreibt die Wafer-Ausbeute produzierte Wafer, welche vollständig unbrauchbar sind. Defekte pro Fläche stellt dabei das Maß für zufällige und Herstellungsfehler dar. Zusätzlich beschreibt α die Komplexität des Herstellungsprozesses. Die Berechnung der Chip-Ausbeute mit Gleichung 2.18 berücksichtigt nicht die Möglichkeit der Kompensation von Fertigungsfehlern durch Maßnahmen der Fehlertoleranz.

Die Autoren in [NK03] vergleichen die Ausbeute in Bezug auf funktionierende Chips pro Wafer für Schaltungen mit und ohne Fehlertoleranz. Dabei bedienen sie sich der Gleichungen 2.17 und 2.18. In dem von ihnen vorgestellten Modell wird allerdings vereinfachend angenommen, dass alle Fehler in Chips mit Maßnahmen der Fehlertoleranz kompensierbar sind. Somit entspricht die Anzahl funktionierender Chips pro Wafer der Anzahl aus Gleichung 2.17.

Die Verwendung von Fehlertoleranz zur Kompensation von Herstellungsfehlern führt dazu, dass ICs mit unterschiedlichen Eigenschaften bezüglich der noch zur Verfügung stehenden Maßnahmen der Fehlertoleranz entstehen. Allen fehlerfreien ICs stehen die zusätzlichen Maßnahmen der Fehlertoleranz weiter zur Verfügung, während ICs mit Fehlern, welche durch die Fehlertoleranz kompensiert wurden, diese Eigenschaft zum Teil oder sogar vollständig nicht mehr aufweisen. Die Ausbeute solcher Produktionen muss dann differenziert betrachtet werden, welches die Autoren in [SKMB03] durch die Einführung einer weiteren Metrik erreichen. Die von ihnen vorgeschlagene Metrik, bezeichnet als PAY (*performance averaged yield*), erlaubt die Unterteilung der Ausbeute für voll funktionale ICs (ohne Fertigungsfehler) und für ICs, welche die Eigenschaften der Fehlertoleranz zu Gunsten der Kompensation von Fertigungsfehlern verloren haben.

Um eine genaue Berechnung der Produktionsausbeute zu ermöglichen, benötigt man in jedem Fall genaue Angaben zu Häufigkeit und Verteilung von Fehlern über den gesamten Wafer. Diese basieren in der Regel auf Erfahrungswerten aus bereits etablierten Produktionsprozessen. Bei einem Umstieg auf andere Fertigungsverfahren mit geringeren Strukturgrößen sind diese Erfahrungswerte noch nicht vorhanden und müssen approximiert werden.

2.3 Fehlerursachen und Auswirkungen

In diesem Abschnitt sollen die Ursachen und Auswirkungen verschiedener Fehler vorgestellt und erläutert werden. Dazu werden zuerst mögliche Klassen von Fehlern nach ihren Ursachen vorgestellt. Diese Unterteilung stellt einen detaillierten Einblick in die unterschiedlichen Ursachen für Fehler in integrierten Schaltungen dar. Mögliche Fehler der Spezifikation und des Entwurfs sind nicht Bestandteil dieser Arbeit und werden aus diesem Grund nicht weiter betrachtet. Im Abschnitt 2.3.4 folgt die Abstraktion aller Fehler nach ihren zeitlichen Eigenschaften.

Der Autor in [Bor05] unterteilt mögliche Fehlerursachen integrierter Schaltungen in statische Variation und dynamische Variation. Die Argumentation in [Bor05] geht von einer steigenden Anzahl von Transistoren aus, bei denen viele Transistoren durch statische Variation nicht benutzbar sein werden. Zusätzlich wird der gesamte Chip durch dynamische Variation altern, was wiederum zu weniger funktionierenden Transistoren im laufenden Betrieb führt. Der Autor sieht damit unweigerlich einen Paradigmenwechsel voraus, der in allen Aspekten des VLSI[6] Entwurfes erfolgen muss. Das betrifft zusätzlich zum Entwurf auch die Architekturen, den Test, die Software und den Herstellungsprozess von integrierten Schaltungen.

Zusätzlich zur statischen und dynamischen Variation von Halbleitern sind es laut dem Autor in [Ala08] auch zufällige katastrophale Fehler, die keine allmähliche Alterung verursachen, sondern einen sofortigen Ausfall zur Folge haben können.

Damit lassen sich die unterschiedlichen Klassen für Fehlerursachen in integrierten Schaltungen folgendermaßen zusammenfassen:

- Statische Variation (*time-zero process variation*)
- Dynamische Variation (*time-dependent device degradation*)
- Zufällige Fehler (*random-faults*)

[6] very-large-scale integration - Integration einer Vielzahl von Transistoren auf einem Chip

2.3.1 Statische Variation

Statische Variation ist wohl die am längsten bekannte Ursache für Fehler in integrierten Schaltungen überhaupt. Diese Ursache ist nicht nur begrenzt auf die Produktion von integrierten Schaltungen, sondern ist als Produktionsfehler im allgemeinen auch in vielen anderen Bereichen ein Begriff. Durch die hohe Komplexität der Halbleiterfertigung und der wenigen und beschränkten Möglichkeiten der Kompensation von Produktionsfehlern wiegt diese Ursache hier besonders schwer.

In [ZGVM04, Bor05, Ala08, WWW06] sind einige Beispiele für statische Variation beschrieben. Dazu zählen unter anderen:

- Zufällige Fluktuationen in der Dotierung von Halbleitern
- Probleme bei der Lithographie
- Variationen bei Ätzverfahren, Oxidation, Abscheidung und Metallisierung
- Verschmutzungen durch äußere Einflüsse

Für den Hersteller ist es von Bedeutung, ob es sich dabei um eine generelle Variation des Herstellungsprozesses oder um einen zufälligen Mangel handelt, da er erstere durch Verbesserungen des Prozesses für folgende Produktionen beseitigen oder zumindest verringern kann.

Die Variation von Fertigungsprozessen stellt dennoch den Normalfall dar. Trotz stetiger technischer Neuerungen in den vielen unterschiedlichen Verfahren der IC Herstellung wird es nicht gelingen, auch nur zwei Transistoren mit identischen physikalischen Eigenschaften zu produzieren. Das bedeutet, integrierte Schaltungen unterliegen immer einer statischen Variation. Demzufolge stellt sich nicht die Frage ob, sondern nur wie stark beeinflussen diese Variationen die Funktionsweise der produzierten Schaltungen.

Im ungünstigsten Fall sind es sogenannte *killer-defects*, das sind zum Beispiel Verunreinigungen oder sehr starke parametrische Schwankungen, welche dazu führen, dass die gewünschte Funktionalität durch den IC nicht erbracht werden kann. Dadurch sinkt die Ausbeute der Produktion und treibt die Herstellungskosten in die Höhe.

Ein weiterer Fall beschreibt parametrische Schwankungen, die zwar zur Nichteinhaltung der Spezifikation führen, aber dennoch sind die produzierten Schaltungen eingeschränkt funktionstüchtig (z.B. bei niedriger Frequenz oder bei erhöhter

Spannung). Damit diese die Ausbeute nicht zu stark beeinflussen, können diese als abgewertete Produkte dennoch verkauft werden. Als bekanntestes Beispiel sei hier der 80486 SX von Intel erwähnt, der trotz defekter FPU[7] den Weg zum Verbraucher fand [Sin02]. Die Ursache für die defekte FPU Einheit war in diesem Fall allerdings keine parametrische Schwankung im Herstellungsprozesses, sondern mindestens ein Design Fehler. Diese Herangehensweise ist für heutige Multi-Core Architekturen nicht mehr wegzudenken [SKMB03, HX10b].

Ebenso kommt es vor das Schaltungen, die nach der Produktion zwar der Spezifikation entsprechen und keine *killer-defects* aufweisen, dennoch kurz nach ihrem Einsatz ausfallen oder Fehler produzieren. Diese Frühausfälle auch bekannt als ELF (*early-life failure*) bleiben, trotz ihres Auftretens im Feld, ein Produktionsproblem und zählen somit zur statischen Variation. Dem Effekt der ELF wird mit Hilfe von *burn-in* Verfahren begegnet. *Burn-in* ist Teil des Produktionstests und setzt alle Schaltungen gezieltem Stress aus, damit mögliche Kandidaten für ELF nicht den Weg zum Nutzer finden. Das grundlegende Problem von *burn-in* Verfahren besteht in der Tatsache, dass alle Schaltungen gezielt zum Altern gebracht werden. Das betrifft dann nicht nur ELF Kandidaten, sondern auch alle anderen Schaltungen und kann auch weitere ELF Kandidaten hervorrufen [Whi10]. Aus diesem Grund wird *burn-in* zunehmend kritisch betrachtet. In [Man08] werden unterschiedliche Möglichkeiten des Wafer-Tests diskutiert, die die Zuverlässigkeit von Schaltungen erhöhen können und gleichzeitig den Aufwand für *burn-in* reduzieren oder *burn-in* Verfahren gänzlich obsolet werden lassen.

Zusammenfassung statische Variation

Zusammenfassend kann man sagen, dass allein die Auswirkungen statischer Variationen sehr unterschiedlich sind. Dies gilt vor allem für kleinere Fertigungsgrößen, bei denen es zu größeren Variationen zwischen einzelnen Wafern, einzelnen Chips bis hin zu Variationen auf einem Chip kommt [SKSH02]. Zusätzlich zum steigenden Einfluss der statischen Variation kommt hinzu das etablierte Testverfahren wie *burn-in* einen negativen Einfluss auf die Integrität der Schaltung bewirken können. Damit ist und bleibt die statische Variation ein ernsthaftes Problem in der Halbleiterindustrie.

[7] floating point unit - bezeichnet eine Einheit in Prozessoren welche Gleitkomma Zahlen verarbeitet

2.3.2 Dynamische Variation

Dynamische Variation bezeichnet die stetige Veränderung von Schaltungsparametern durch unterschiedliche Effekte, die in diesem Abschnitt vorgestellt werden. Die dynamische Variation ist auch unter dem Begriff Alterung bekannt. Die Ursache für die Alterung von integrierten Schaltungen wird sowohl durch interne als auch durch externe Einflüsse hervorgerufen. Grundsätzlich unterliegt jede Schaltung einem Alterungsprozess und wird zu einem bestimmten Zeitpunkt ihre Spezifikation nicht mehr erfüllen. Bis zum Ende des letzten Jahrhunderts waren diese Alterungseffekte eher unbedeutend, da sie erst nach 40 Jahren Einsatzzeit relevant wurden [KKC98]. Die meisten Systeme wurden und werden allerdings viel zeitiger durch neue ersetzt. Demzufolge konnte diesen Alterungseffekten lange Zeit wenig Bedeutung zugemessen werden. Durch stetige Weiterentwicklungen von Herstellungsprozessen zur Leistungssteigerung von ICs sind Alterungseffekte in heutigen Technologien ein relevantes Problem [AM05, SABR04b].

Ein sehr ausführlicher Überblick über unterschiedliche Alterungseffekte und ihre Ursachen ist in [Jed06, Sri06] zu finden. Dennoch werden an dieser Stelle einige relevante Fehlereffekte vorgestellt.

Elektromigration (EM)

Elektromigration bezeichnet den Effekt des Transportes von Ionen in einem elektrischen Leiter. Dieser Transport wird durch eine hohe Stromdichte verursacht und ist seit über 100 Jahren bekannt [AV10]. Seit Mitte der 60er Jahre ist Elektromigration als mögliche Ursache für Alterungseffekte auch in integrierten Schaltungen bekannt [Bla69]. Dieser Effekt wird als Hauptursache für Fehler in metallischen Leitungen und den sogenannten Vias, den Durchkontaktierungen zwischen verschiedenen metallischen Leitungsebenen, angesehen. Gleichung 2.19 [AVU+08] beschreibt EM wie folgt:

$$MTTF_{EM} = A \cdot j^{-n} \cdot e^{\frac{Q}{k \cdot T}} \tag{2.19}$$

Hier stellt A eine technologische Konstante dar, welche empirisch zu ermitteln ist. Die Stromdichte der Verbindungsleitung ist j, n ist eine Konstante für das Material der Leitung, Q ist die Aktivierungsenergie für EM, k ist die Blotzmann Konstante und T die Temperatur in Kelvin.

Abbildung 2.6 zeigt eine Fotographie einer Verbindungsleitung aus Aluminium mit Elektromigration. Durch den Abtransport von Ionen im Leiter erhöht

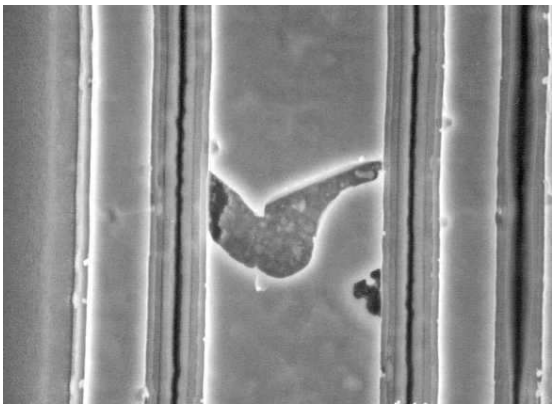


Abbildung 2.6: Elektromigration einer Verbindungsleitung aus Aluminium (entnommen aus [Jed06])

sich die Stromdichte an dieser Stelle und das erhöht wiederum die Temperatur an dieser Stelle. Eine höhere Temperatur begünstigt dann wieder den Abtransport von Ionen. Auf das Schaltverhalten bezogen verlängert sich, während der Ausbildung der Störstelle, die Zeit zum Umladen der jeweiligen Kapazität. Die zeitliche Verzögerung des Signals auf diesem Leiter steigt weiter an und manifestiert sich letztlich als *open fault*. Durch eine Umkehr der Stromrichtung kann dieser Effekt teilweise rückgängig gemacht werden [AVU+08].

Stressmigration (SM)

Bei der Stressmigration werden, wie bei der Elektromigration, metallische Atome aus den Verbindungsleitungen gelöst und wandern ab. Dieser Effekt wird durch unterschiedliche thermische Ausdehnungseigenschaften der verwendeten Materialien bewirkt. Gleichung 2.20 [Jed06] beschreibt diesen Effekt:

$$MTTF_{SM} = B_0(T_0 - T)^{-n} e^{\frac{E_a}{kT}} \tag{2.20}$$

Hier sind B_0 und n materialabhängige Konstanten und k die Boltzmann Konstante. Die Temperatur für den stressfreien Zustand des Metalls bezeichnet T_0. E_a beschreibt die elektrische Feldstärke. Abbildung 2.7 beinhaltet eine Fotographie welche mögliche Auswirkungen von SM zeigt.

Stressmigration führt wie Elektromigration zu einer stetigen Verschlechterung des Schaltverhaltens, welches sich anschließend in einem *open fault* manifestiert.

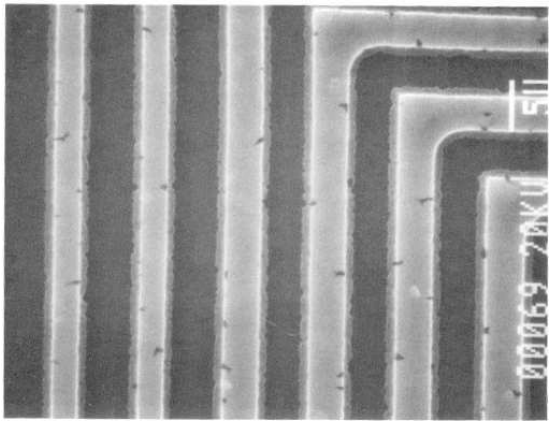


Abbildung 2.7: Stressmigration bei Metallleitungen (entnommen aus [Jed06])

Time dependent dielectric breakdown (TDDB)

Ein ebenfalls sehr bekannter und ausführlich untersuchter Fehlereffekt ist TDDB, der vor allem in sehr dünnen SiO_2 und $SiON$ Dielektrika auftritt. Dieser Effekt lässt sich nach [Nic07] in mehrere Phasen unterteilen:

1. Ausbildung von Störstellen
2. *Soft-Breakdown* (SBD)
3. *Post-Breakdown* (PBD)
4. *Hard-Breakdown* (HBD)

In der ersten Phase enstehen durch Stress immer mehr Störstellen im Dielektrikum. Anschließend kommt es zum SBD, dieser zeichnet sich durch einen abrupten Anstieg vom *gate*-Strom aus. In der PBD Phase verschlechtert sich der Zustand des Dielektrikums weiter bis es zum HBD kommt. Dieser bezeichnet einen Kurzschluss zwischen *gate-source*, *gate-drain* oder *gate-channel* [YHC11]. Bei höheren Spannungen verringert sich die Zeit zwischen SBD und HBD was dazu führt, dass der erste *breakdown* dann meistens ein HBD ist [Nic07].

Eine mögliche Modellierung dieses Effektes ist in Gleichung 2.21 [Jed06] gegeben.

$$MTTF_{TDDB} = \tau_o(T) \cdot e^{\frac{G(T)}{E_{ox}}} \tag{2.21}$$

Hier beschreiben $\tau_o(T)$ und $G(T)$ temperaturabhängige Faktoren und E_{ox} das Elektrische Feld am Dielektrikum. In Abbildung 2.8 ist die Folge dieses Effektes anhand einer Fotografie zu sehen.

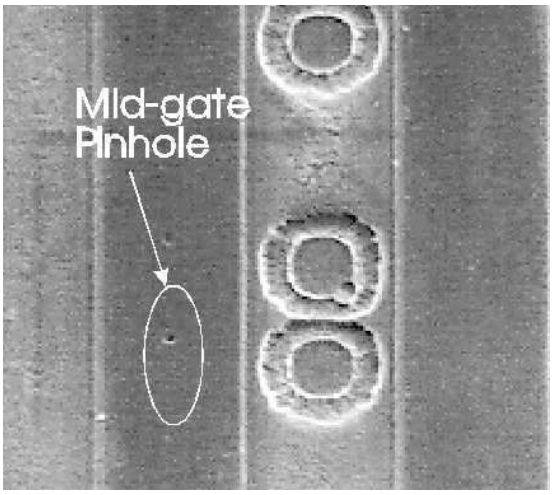


Abbildung 2.8: TDDB *breakdown* im Dielektrikum einer Transistorbasis (entnommen aus [Jed06])

Trotz vieler Untersuchungen dieses Effektes sind die genauen Ursachen noch nicht vollständig geklärt. Aus diesem Grund werden die unterschiedlichen Modelle, und die Daten aus denen sie erstellt werden, kontrovers diskutiert [Sta02].

Negative bias temperature instability (NBTI)

Der NBTI Effekt ist seit den 70er Jahren ein bekanntes Phänomen, welches für instabile Parameter bei PMOS Transistoren sorgt [AM05, Ala08, SB03]. Dieser Effekt tritt in PMOS Strukturen mit negativer Basisspannung auf und sorgt ebenfalls für parametrische Veränderungen der Transistoren. Ein typischer Effekt ist das Ansteigen der Schwellenspannung, was zu einem verzögerten Umschalten und damit zu langsameren Schaltverhalten führt. Durch Simulationen eines analytischen Modells wurde eine Erhöhung der Schwellenspannung um $25\% - 30\%$ in einem Zeitraum von ca. 10 Jahren ermittelt [KKS06]. Gleichung 2.22 [Sri06] zeigt eine mögliche Modellierung dieses Effektes:

$$MTTF_{NBTI} = [(ln(\frac{A}{1+2e^{\frac{B}{kT}}}) - ln(\frac{A}{1+2e^{\frac{B}{kT}}}C)) \times \frac{T}{e^{\frac{-D}{kT}}}]^{\frac{1}{\beta}} \tag{2.22}$$

Die Schaltungseigenschaften einiger von NBTI betroffenen Transistoren können sich durch den Wegfall der negativen Spannungen an der Basis wieder verbessern [AM05, SB03]. Ebenfalls gibt es Entwicklungen, die versuchen diesen Effekt durch geschickte Schaltungssynthese zu reduzieren [KKS07].

Thermal cycling (TC)

Ein weiterer Fehlereffekt der dynamische Variation verursacht, wird als *thermal cycling* (TC) bezeichnet. Im Gegensatz zu SM beschreibt TC nicht den Abtransport von Atomen durch verschiedene Ausdehnungseigenschaften der Materialien, sondern die Materialermüdung die beispielsweise beim Ein-/Ausschalten oder auch durch unterschiedliche Belastungen oder Stromsparmodi auftreten können. Für hohe Frequenzen von thermalen Zyklen gibt es bislang kein validiertes Modell. Allerdings kann man diesen Effekt für niedrige Frequenzen durch Gleichung 2.23 [Sri06] beschreiben:

$$MTTF_{TC} = (\frac{1}{T - T_{env}})^q \tag{2.23}$$

Hier beschreibt q den Coffin-Manson Exponenten, einen materialabhängigen Faktor, und $T - T_{env}$ den Temperaturunterschied von Schaltung und Umgebung. Abbildung 2.9 beinhaltet eine Fotographie eines zerstörten Chips durch *thermal cycling*.

Hot carrier injection (HCI)

Bei der *hot carrier injection* werden hochenergetische Elektronen vom Halbleiter in die angrenzenden Dielektrika injiziert. Diese *hot carrier* ("heiße Elektronen") sorgen für eine stetige Veränderung der Charakteristika der betroffenen Transistoren und können zu einem permanenten Fehlverhalten der Schaltung führen [Pag03, LLY+01]. Durch Gleichung 2.24 [Jed06] kann der Einfluß von HCI auf die $MTTF$ einer Schaltung beschrieben werden:

$$MTTF_{HCI} = B(I_{sub|gate})^N e^{\frac{E_a}{kT}} \tag{2.24}$$

Hier beschreibt $B(I)^N$ einen technologieabhängigen Faktor, wobei I_{sub} der Maximalstrom des Substrates für n-Kanal und I_{gate} der Maximalstrom am *gate* für p-Kanal Transistoren während des Stresses darstellt. E_a ist die elektrische Feldstärke, k die Boltzmann Konstante und T die Temperatur.

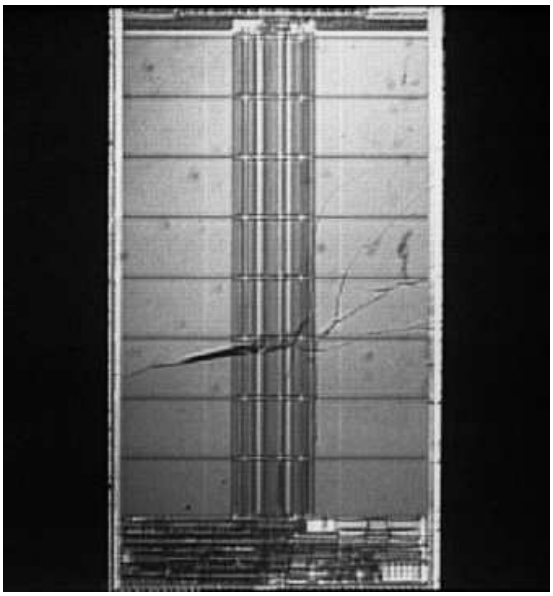


Abbildung 2.9: Fotographie eines gesprungenen Chips durch *thermal cycling* (entnommen aus [Jed06])

Zusammenfassung dynamische Variation

Durch die zunehmende Verkleinerung der Strukturen von integrierten Schaltungen steigert sich deren Anfälligkeit auch gegenüber Alterungseffekten. Einige der vorgestellten Fehlereffekte sind seit langem bekannt, allerdings waren deren Auswirkungen lange Zeit sehr gering und damit nicht relevant für die Halbleiterfertigung. Durch die stetige Skalierung und die damit verbundenen Änderungen der physikalischen Parameter erhöhte sich der Einfluss dieser Fehlereffekte. Mittlerweile sind Alterungseffekte ein ernstzunehmendes Problem für die Zuverlässigkeit und Lebensdauer von ICs.

2.3.3 Zufällige Fehler

Diese dritte Klasse von Fehlern bezeichnet vor allem Störungen des Systems, die durch radioaktive Strahlung oder auch elektromagnetische Interferenzen hervorgerufen werden. Dabei können die Ursachen sowohl äußerer als auch innerer Natur sein. Laut den Autoren in [DNR02] gibt es drei wesentliche Gründe für Strahlungen, die integrierte Schaltungen betreffen und Fehler verursachen können.

- Neutronen mit geringer Energie - Diese stammen entweder aus dem Weltall oder aus radioaktiven Zerfallsprozessen von Materialien auf der Erde und reagieren mit einem Bor-Isotop, welches in integrierten Schaltungen vorhanden sein kann. Sie können außerdem durch nukleare Reaktionen energetische Alphateilchen freisetzen.
- Neutronen mit hoher Energie - Diese stammen meistens aus dem Weltall und sind in der Lage dicke Betonwände zu durchdringen. Durch eine Kollision mit Silizium werden weitere Ladungsträger frei.
- Intern erzeugte Partikel - Durch radioaktiv verunreinigtes Halbleitermaterial oder umgebendes Packmaterial, beispielsweise Blei, können ebenso Alphateilchen freigesetzt werden.

Zufällige Fehler gewinnen zunehmend an Bedeutung, da durch sinkende Strukturgrößen die Anfälligkeit der Schaltungen gegenüber diesen Effekten zunimmt [DNR02, Con03, TIC+05]. Die meisten dieser Effekte sorgen für ein ungewolltes Umladen von Kapazitäten und werden als sogenannte SET (*single event transient*) bezeichnet. Durch das Umladen einer Speicherzelle beziehungsweise durch das Abspeichern eines SET werden SEU (*single event upset*) produziert. Sind mehrere Speicherzellen betroffen, spricht man von MEU (*multiple event upsets*). Diese Effekte sind nur temporär und beeinträchtigen den physischen Zustand der Schaltung nicht. Allerdings gibt es auch Effekte, die dauerhaften Schaden an der Schaltung verursachen. Der SEL (*single event latchup*) beschreibt einen Kurzschluss der in CMOS implizit vorhandenen parasitären Bipolartransistoren, verursacht durch Spannungsspitzen, elektromagnetische Interferenzen oder auch Neutronenstrahlung, der die Funktion der Schaltung vollständig zerstören kann [Nic06]. Der SEB (*single event burnout*) ist ein Effekt der vor allem in Leistungshalbleitern zum Tragen kommt. SEB wird durch kosmische Strahlung hervorgerufen und kann permanenten Schaden an diesen Halbleiterelementen hervorrufen [Dod07]. Ein weiterer zufälliger Fehler, der permanenten Schaden verursachen kann, wird als SEGR (*single event gate rupture*) bezeichnet. Bei diesem

Effekt sorgen schwere hoch energetische Atome, die einen MOSFET[8] durchqueren, für einen katastrophalen Kurzschluss [PTB+10]. Für diesen Effekt ist allerdings noch unklar, ob nicht noch weitere Vorbedingungen erfüllt sein müssen, damit es zu einem fatalen Kurzschluss kommt [PTB+10].

2.3.4 Zeitliche Eigenschaften von Fehlern

Die Unterteilung der Fehler nach ihren Ursachen ist immer dann von hoher Bedeutung, wenn durch einzelne Maßnahmen gezielt das Auftreten eines speziellen Fehlereffektes verhindert werden soll. Erweist sich eine spezielle Ursache beispielsweise während der Fertigung oder im Betrieb als besonders dominant, dann wird vorrangig der Fertigungsprozess angepasst, um die Ursache abzuschwächen oder vollständig zu beseitigen.

In diesem Abschnitt folgt eine Unterteilung der Fehler nach ihren Eigenschaften. Ist ein Fehler erst einmal existent, dann ist dessen eigentliche Ursache zur Behandlung in der Regel nicht mehr relevant. Für die Behandlung eines Fehlers sind dann andere Eigenschaften von größerer Bedeutung. Dabei ist die Dauer ihres Auftretens eine wichtige Eigenschaft und daher ist die folgende Unterteilung in transiente, intermittierende und permanente Fehler gängig [Con03, KK07].

Ein transienter Fehler bezeichnet eine zufällige und temporäre Störung des Systems. Durch diesen Fehler wird die zugrunde liegende Hardware nicht verändert und damit entsteht kein dauerhafter Schaden. Die Ursache transienter Fehler sind die in Absatz 2.3.3 vorgestellten zufälligen Fehler, welche ein ungewolltes Umladen von Signalpegeln bewirken und zu keinem permanenten Schaden führen.

Intermittierende und permanente Fehler sind Störungen des Systems, deren Ursache auf eine Variation der Hardware zurückzuführen ist. Im Gegensatz zu permanenten Fehlern müssen für intermittierende Fehler allerdings weitere Randbedingungen erfüllt sein. Das kann beispielsweise eine bestimmte Temperatur oder das Über- beziehungsweise Unterschreiten eines bestimmten Niveaus der Versorgungsspannung sein. Bei einem permanenten Fehler ist die Variation der Hardware so gravierend, dass ein ständiges Fehlverhalten daraus resultiert. Die vorgestellten Fehlerursachen aus Abschnitt 2.3.1 und 2.3.2 zur statischen und dynamischen Variation beschreiben die Ursachen für beide Fehler. Viele der vorgestellte Ursachen von Fehlern aus dem Abschnitt 2.3.2 können zu intermittierenden Fehlern führen, die sich bei fortwährender Verschlechterung der Schaltungsparameter zu permanenten Fehlern ausweiten.

[8] metal oxid semiconductor field effect transistor - Feldeffekttransistor mit isoliertem Gate

2.3.5 Diskussion

In diesem Absatz wird die Relevanz der verschiedenen Ursachen für Fehler diskutiert. In der Literatur finden sich dazu die unterschiedlichsten Erklärungen für die Dominanz bestimmter Fehlereffekte. In [SABR04a] wird argumentiert, dass nur Alterungseffekte in der letzten Phase der Badewannenkurve relevant sind, da alle ELF durch Maßnahmen wie *burn-in* irrelevant werden. Gegen diese Argumentation sprechen zum Einen die sinkende Wirksamkeit von *burn-in* und I_{DDQ}[9] Tests aufgrund geringerer Versorgungsspannungen und hoher Leckströme [RMC06]. Ebenfalls können *burn-in* Verfahren gleichzeitig auch die Schaltungsparameter verschlechtern und zu vorzeitiger Alterung führen [Whi10, ITR11]. Zum Anderen werden Fehler in der mittleren Phase der Badewannenkurve gänzlich vernachlässigt. Diese sind laut [KKC98] vor allem auf unsachgemäße Handhabung und Umwelteinflüsse zurückzuführen.

Diese unterschiedlichen Argumentationen zeigen, dass zum jetzigen Zeitpunkt noch nicht klar ist, welche Probleme zukünftig die größte Relevanz für integrierte Schaltungen haben. Sobald sich eine bestimmte Ursache als besonders dominant herausstellt, müssen gezielt Lösungen gefunden werden. Allerdings ist man sich heute bei einigen Fehlereffekten über deren genaue Ursache noch nicht im Klaren. Ferner ist nicht ausgeschlossen, dass durch weitere Skalierungen der Fertigungstechnologien neue Fehlereffekte auftreten können. Weiterhin sind mögliche Wechselwirkungen zwischen unterschiedlichen Fehlereffekten nicht ausgeschlossen und können zu einer Verstärkung aber auch einer Abschwächung einzelner Fehlerursachen führen.

Aus diesem Grund wird auch in Zukunft die Behandlung der Auswirkungen von Fehlern durch Maßnahmen der Fehlertoleranz unabdingbar bleiben. Daher sollten zukünftige Lösungen flexibel sein, um auf Auswirkungen von Fehlern während ihrer gesamten Einsatzzeit und direkt nach ihrer Produktion reagieren zu können.

[9] I_{DD} quiescent - bezeichnet einen Teil des Fertigungstests bei dem der Ruhestrom einer Schaltung gemessen wird

KAPITEL 3

Stand der Technik

In diesem Kapitel wird der Stand der Technik für Maßnahmen der Fehlertoleranz, speziell für permanente Fehler, vorgestellt und erläutert. Zum anderen wird auf Verfahren eingegangen, welche gezielt auf die Abschwächung oder Vermeidung der Fehlerursache abzielen. Wie bereits das letzte Kapitel bezieht sich auch hier der Stand der Technik auf Fehler deren Ursachen in Produktion und Betrieb zu finden sind. Mögliche Verfahren zur Behandlung oder Vermeidung von Spezifikations- oder Entwurfsfehlern sind nicht Bestandteil dieser Arbeit.

Im ersten Abschnitt liegt der Fokus auf Maßnahmen der Fehlertoleranz, welche eine Kompensation für permanente Fehler liefern. Der zweite Abschnitt beschreibt anschließend die Möglichkeiten des *variation aware design*. Im Gegensatz zu den Maßnahmen der Fehlertoleranz, welche auf bestimmte Eigenschaften von Fehlern reagieren, zielt das *variation aware design* auf die Behebung oder Abschwächung der eigentlichen Fehlerursache ab.

3.1 Fehlertoleranz für permanente Fehler

Mit der Entstehung von elektronischen Rechensystemen entstand auch der Forschungszweig der Fehlertoleranz für diese Systeme. Ursache dafür waren die katastrophalen Ausfallraten der verwendeten Komponenten wie Elektronenröhren und die steigende Komplexität dieser Systeme. Seit dieser Zeit ist bekannt, dass man zuverlässige Systeme aus unzuverlässigen Subsystemen erstellen kann, jedoch mit den damit verbundenen Kosten von größeren Schaltungen und dem daraus resultierenden höheren Stromverbrauch und/oder zusätzlich benötigter Rechenzeit [vN56].

Die gängige Unterteilung in der Literatur [Pra96,KK07,Lal01] für Maßnahmen zur Fehlertoleranz ist folgende:

- Hardware-Redundanz
- Informations-Redundanz
- Zeit-Redundanz
- Software-Redundanz

Die Unterteilung in diese vier Kategorien von Redundanz-Strategien ist nicht für alle Maßnahmen der Fehlertoleranz praktikabel, weil sie sich nicht eindeutig in eine dieser Kategorien einteilen lassen. Deshalb wird häufig noch von hybrider Redundanz gesprochen. Diese bezeichnet Maßnahmen, welche eine Fehlertoleranz aus einer Kombination verschiedener Redundanz-Strategien erreicht.

Der Einsatz einer Maßnahme der Fehlertoleranz hängt ganz entscheidend davon ab, wie schnell das System auf welche Fehler reagieren soll. Gerade hier spielt das zeitliche Verhalten der Fehler eine wichtige Rolle. Im Gegensatz zu permanenten Fehlern müssen Maßnahmen der Fehlertoleranz für intermittierende und transiente Fehler die Systemfunktion im laufenden Betrieb überwachen. Nur dann können diese unregelmäßig beziehungsweise zufällig auftretenden Fehler überhaupt erkannt und kompensiert werden. Soll eine Kompensation von transienten oder intermittierenden Fehlern mit ihrem Auftreten erfolgen, dann eignet sich eine solches Verfahren oft auch für permanente Fehler.

Zeit-Redundanz und Software-Redundanz werden im folgenden nicht weiter betrachtet. Maßnahmen der Zeit-Redundanz basieren im Wesentlichen auf einer Wiederholung der Berechnung oder Übertragung und sind damit keine Möglichkeit der Fehlertoleranz für permanente Fehler. Software-Redundanz bezieht sich auf die Kompensation von Fehlern in Software und ist somit ebenfalls nicht relevant für diese Arbeit.

Zur Realisierung der Fehlertoleranz sind laut [Sie91] bis zu 10 Stufen zu durchlaufen. Welche Stufen das jeweils sind ist abhängig von der gewählten Maßnahme. In Tabelle 3.1 werden die einzelnen Stufen vorgestellt und kurz erläutert.

fault confinement	Beschreibt das Verringern der Verbreitung von auftretenden Fehlern.
fault detection	Beschreibt das Erkennen von auftretenden Fehlern.
fault masking	Beschreibt das Maskieren von auftretenden Fehlern.
retry	Beschreibt das wiederholte Ausführen
diagnosis	Beschreibt das Lokalisieren des aufgetretenen Fehlers.
reconfiguration	Beschreibt das Ersetzen oder Isolieren von fehlerhaften Komponenten.
recovery	Beschreibt das Eliminieren des durch einen Fehler verursachten Effektes.
restart	Beschreibt einen Neustart im Falle einer fehlgeschlagenen bzw. nicht vorhandenen Korrekturmaßnahme.
repair	Beschreibt den physischen Austausch defekter Komponenten.
reintegration	Beschreibt die Integration der physisch reparierten Komponenten.

Tabelle 3.1: Stufen zum Erreichen von Fehlertoleranz (vgl. [Sie91])

Zum Erreichen der Fehlertoleranz eines Systems sind mehrere oder sogar alle Stufen zu kombinieren. In den folgenden Abschnitten wird auf Maßnahmen der Fehlertoleranz eingegangen, welche eine Kompensation von permanenten Fehlern leisten. Dabei sind die Maßnahmen in den Abschnitten 3.1.1, 3.1.2 und 3.1.3 zusätzlich in der Lage auch transiente und intermittierende Fehler zu kompensieren. Im danach folgenden Abschnitt 3.1.4 wird detailliert auf die einzelnen Schritte der aktiven Hardware-Redundanz eingegangen, welche transiente und intermittierende Fehler nicht berücksichtigt.

3.1.1 Passive Hardware-Redundanz

Fehlertoleranz durch passive Hardware-Redundanz beschreibt Techniken, welche auftretende Fehler durch eine Fehlermaskierung kompensieren. Das bekannteste Beispiel von passiver oder statischer Hardware-Redundanz ist TMR (*triple modular redundancy*). Diese Technik wurde ursprünglich durch von Neumann [vN56] vorgestellt und ist in Abbildung 3.1 dargestellt.

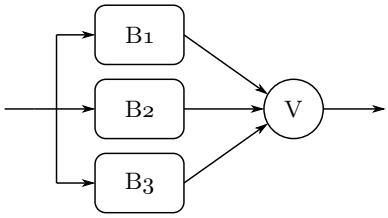


Abbildung 3.1: Schematische Darstellung eines TMR-Systems

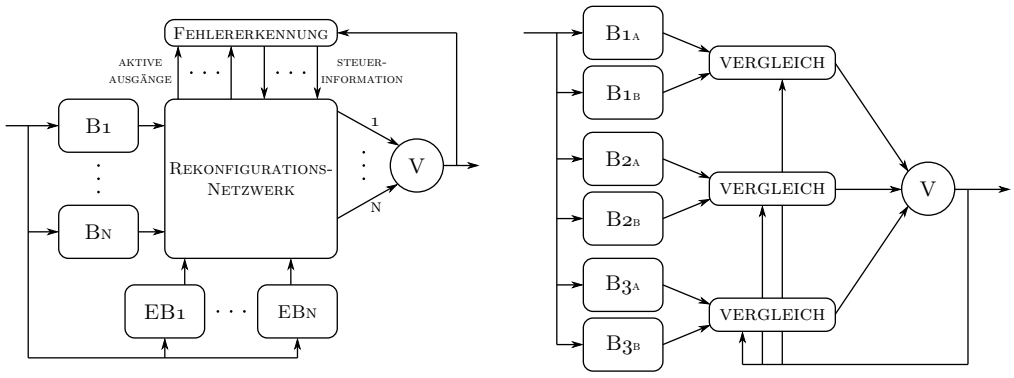
Diese Technik verdreifacht das ursprüngliche System und führt alle Ausgänge zu einer Voter Einheit, die dann durch eine Mehrheitsentscheidung eine Ausgabe produziert. Ein TMR-System ist in der Lage, einen Fehler in einer beliebigen Komponente zu kompensieren, solange der Voter nicht betroffen ist. Durch beliebige Erweiterungen können sogenannte NMR-Systeme (N *modular redundancy*) erstellt werden. Diese können dann genau n Fehler kompensieren, wobei $n = (N - 1)/2$ ist. Durch NMR-Systeme können sowohl transiente, intermittierende als auch permanente Fehler mit ihrem Auftreten kompensiert werden. Die Fehler werden nicht explizit behandelt, sondern einfach maskiert. Allerdings bedeutet der Einsatz von NMR einen erheblichen Mehraufwand von Fläche und eine deutliche Steigerung der Verlustleistung. Zusätzlich zur Fähigkeit, Fehler im laufenden Betrieb zu maskieren, kann TMR auch eingesetzt werden, um eine Steigerung der Ausbeute zu bewirken [VVB+09, NK03].

TMR Techniken sind trotz ihres erheblichen Mehraufwandes an Fläche und Verlustleistung ein probates Mittel zur Steigerung der Zuverlässigkeit. Jedoch kann TMR nicht eingesetzt werden, um die Lebensdauer des Systems zu erhöhen [KK07].

Zur Reduzierung des Aufwands an Hardware und Verlustleistung stellt der Autor in [Aug12] ein Verfahren vor, welches ein TMR-System nur für spezifische Signale einer Komponente erzeugt. Somit wird eine Fehlertoleranz nur für die ausgewählten Signale möglich, welche dann auch mit weniger Fläche auskommt.

3.1.2 Hybride Hardware-Redundanz

Als hybride Hardware-Redundanz werden Systeme bezeichnet, welche eine Maskierung von Fehlern leisten und darüber hinaus fehlerhafte Komponenten für den weiteren Betrieb austauschen beziehungsweise abschalten. Somit vereinen diese Systeme die Eigenschaften von passiver und aktiver Hardware-Redundanz. Verfahren der aktiven Hardware-Redundanz werden später gesondert betrachtet. Grundsätzlich gilt für die Systeme der aktiven Hardware-Redundanz, dass eine Fehlertoleranz durch eine Kombination von Fehlererkennung, Diagnose und Rekonfiguration erreicht wird.



(a) *N*MR System mit Ersatzbaugruppen (b) Triple-Duplex System

Abbildung 3.2: Systeme mit hybrider Hardware-Redundanz

In Abbildung 3.2 sind zwei mögliche Systeme mit hybrider Hardware-Redundanz dargestellt. Abbildung 3.2(a) zeigt ein *N*MR System mit Ersatzbaugruppen. Hier werden Fehler durch die Eigenschaft des passiven *N*MR Systems maskiert und zusätzlich wird durch die Fehlererkennung der Austausch defekter Baugruppen realisiert. Eine weitere Möglichkeit der hybriden Hardware-Redundanz zeigt Abbildung 3.2(b). In dieser Umsetzung wird jede Baugruppe des TMR-Systems verdoppelt. Durch eine Rückkopplung des Voterausganges mit anschließendem Vergleich der verdoppelten Komponenten kann eine fehlerhafte Baugruppe erkannt und für den weiteren Betrieb abgeschaltet werden.

Im Gegensatz zu Systemen der passiven Hardware-Redundanz können hybride Hardware-Redundanz Strategien zur Steigerung der Lebensdauer beitragen. Gleichzeitig stellen solche Systeme die Möglichkeit bereit, sowohl transiente, intermittierende als auch permanente Fehler zu kompensieren. Allerdings ist der Auf-

wand für hybride Hardware-Redundanz stets höher als der von passiver Hardware-Redundanz.

3.1.3 Informations-Redundanz

Techniken, welche eine Fehlertoleranz durch Informations-Redundanz erreichen, basieren auf einer Fehlerkorrektur durch Codes (ECC - *error correcting code*). Im Allgemeinen bezeichnen Codes eine injektive Abbildung von Wörtern eines Alphabets auf Wörter eines anderen Alphabets. Durch eine Abbildungsfunktion (Codierung) können Codewörter erzeugt werden, welche redundante Informationen enthalten. Dadurch wird es möglich, Fehler in Codewörtern zu erkennen und sogar zu korrigieren. Entscheidend ist hier die geringste Hamming-Distanz zwischen zwei Codewörtern. Diese definiert die Anzahl der unterschiedlichen Bitstellen zweier Codewörter eines binären Blockcodes. Die Wahl der Hamming-Distanz für einen Code bestimmt die Anzahl der erkennbaren und korrigierbaren Fehler. Die Fehlererkennung durch EDC (*error detecting code*) reicht nicht aus, um ein fehlertolerantes System umzusetzen. Oft wird eine Fehlererkennung durch Codes in Kombination mit einer erneuten Übertragung (Zeit-Redundanz) genutzt, um eine Fehlertoleranz zu erreichen. Diese funktioniert dann allerdings nicht für permanente Fehler, wenn der gleiche Übertragungsweg oder Speicher gewählt wird.

In fehleranfälligen Übertragungskanälen oder Datenspeichern kommen Codes zur Fehlererkennung beziehungsweise zur Fehlerkorrektur oft zum Einsatz. Dort bieten sie eine günstigere Alternative zur Hardware-Redundanz, also einer Vervielfachung des Übertragungskanals oder des Speichers.

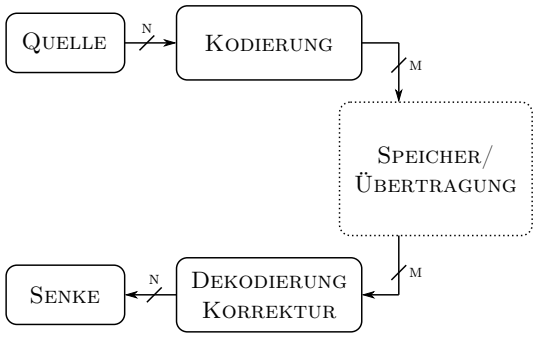


Abbildung 3.3: Schematische Darstellung der Informations-Redundanz zur Kompensation von permanenten Fehlern

In Abbildung 3.3 ist das grundsätzliche Schema der Informations-Redundanz zur Kompensation von Fehlern dargestellt. Ausgehend von einer Quelle wird die Information kodiert. Anschließend wird diese kodierte Information übertragen oder gespeichert. Danach erfolgt eine Dekodierung und eine mögliche Korrektur der Information im Fehlerfall. Die ursprüngliche Information ist dann in der Senke verfügbar. Möglichen Fehlern bei Übertragung oder Speicherung der Information kann somit entgegengewirkt werden.

3.1.4 Aktive Hardware-Redundanz

In den vorherigen Abschnitten wurden Verfahren erläutert, welche eine Fehlertoleranz für transiente, intermittierende und permanente Fehler leisten. Gerade bei diesen Verfahren ist das Bereitstellen der Eigenschaft zur Erkennung und Kompensation von Fehlern im laufenden Betrieb sehr aufwendig. Der Einsatz solcher Verfahren ist nur dann gerechtfertigt, wenn kurzzeitige Störungen oder Abweichungen des Systems inakzeptabel sind und zu einem sofortigen Ausfall führten.

Bei der aktiven Hardware-Redundanz wird vorausgesetzt, dass kurzzeitige Störungen und Abweichungen kein Problem darstellen. Somit kann der sehr hohe Aufwand zur sofortigen Erkennung und Kompensation von Fehlern stark reduziert werden. Im Wesentlichen basieren Systeme der aktiven Hardware-Redundanz auf einer Rekonfiguration des Systems. Dazu werden defekte Baugruppen entweder durch Ersatzbaugruppen ausgetauscht oder im weiteren Betrieb nicht mehr benutzt.

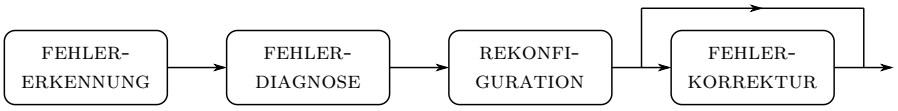


Abbildung 3.4: Phasen der aktiven Hardware-Redundanz

Die Verwendung einer Rekonfiguration erreicht für sich gesehen keine Fehlertoleranz. Vielmehr impliziert eine Rekonfiguration mehrere Schritte (siehe Abbildung 3.4), welche sowohl vor als auch nach dem Rekonfigurieren ablaufen müssen, damit eine Fehlertoleranz erreicht wird. Bevor eine Rekonfiguration stattfinden kann, müssen Fehler zuerst erkannt und anschließend diagnostiziert werden. Je nach Anforderungen an das System kann der Rekonfiguration noch eine Fehlerkorrektur folgen.

In der Literatur wird eine Rekonfiguration des Systems, welche für den Anwender transparent ist, oft auch als BISR (*built-in self-repair*) oder Selbstreparatur bezeichnet [Lal01]. Auch der Begriff *self-healing* wird verwendet und beschreibt letztlich eine Rekonfiguration die dazu führt, dass Teile der Schaltung sich von auftretenden Alterungseffekten erholen können [SBK06, AVU+08, LK02]. Weiterhin leisten nicht alle Verfahren zur Kompensation von permanenten Fehlern eine Fehlerkorrektur im laufenden Betrieb. Vielmehr geht es in vielen Verfahren vor allem darum, ein System mit einem erkannten und diagnostizierten permanenten Fehler durch Rekonfiguration so zu modifizieren, dass dieser Fehler im weiteren Betrieb kein Fehlverhalten mehr verursachen kann.

Im weiteren Verlauf dieses Kapitels wird aus Gründen der Konsistenz ausschließlich der Begriff der Rekonfiguration verwendet, auch wenn die jeweiligen Autoren der zitierten Arbeiten unterschiedliche Begriffe verwenden. In den folgenden Abschnitten wird auf die einzelnen Prozesse von Fehlererkennung, Fehlerdiagnose, Rekonfiguration und Fehlerkorrektur, welche Bestandteil der aktiven Hardware-Redundanz sind, detailliert eingegangen.

3.1.4.1 Fehlererkennung

Bevor es zu einer Rekonfiguration eines Systems kommt, sind mögliche Fehler zu erkennen. Abbildung 3.5 enthält eine Übersicht von Möglichkeiten der Fehlererkennung, welche grundsätzlich für den Einsatz im Feld zur Verfügung stehen und sich damit auch zur Erkennung von Alterungseffekten eignen.

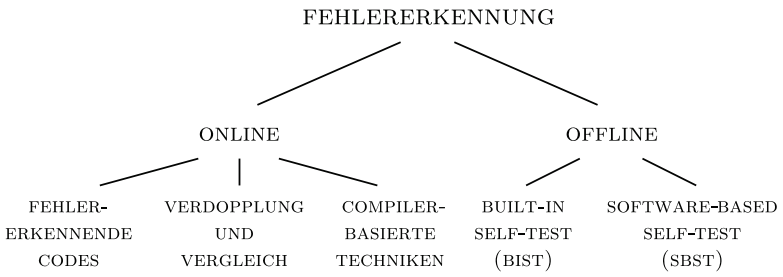


Abbildung 3.5: Möglichkeiten der Fehlererkennung im Feld

Eine Fehlererkennung geschieht entweder *online* oder *offline*. Bei *online*-Verfahren kann zum einen eine Verdopplung mit anschließendem Vergleich oder ein fehlererkennender Code eingesetzt werden. Beide Verfahren sind im Feld anwendbar, da alle notwendigen Strukturen auf dem Chip integriert sind. Allerdings ist

eine Fehlererkennung durch Verdopplung und Vergleich auch mit mindestens einer Verdopplung an Fläche und Leistungsaufnahme verbunden. Fehlererkennung durch Codes ist vor allem zum Absichern von Übertragungen oder Speicherungen eine häufig verwendete Technik. Trotz vorhandener Codes zur Fehlererkennung mit anschließender Korrektur für spezielle Logikschaltungen [DJ08, VS07, NHS01] ist dies keine generelle Möglichkeit zur Fehlererkennung für beliebige Baugruppen von Prozessoren.

Eine weitere Möglichkeit zur *online*-Fehlererkennung besteht durch compilerbasierte Techniken. Dazu fügt der Compiler zur Übersetzungszeit zusätzliche Instruktionen ein, welche dann beispielsweise typische Fehlersymptome oder auch Abweichungen von Invarianten feststellen können [SLR+08]. In [OMM02] wird eine Fehlererkennung durch das Ausführen unterschiedlicher Programme mit identischer Funktionalität und anschließendem Vergleich erreicht. Durch die Verwendung zusätzlicher Instruktionen ermöglichen diese compilerbasierten Verfahren eine *online*-Fehlererkennung. Dies bedeutet allerdings auch eine permanente Mehrbelastung des Systems durch eine gesteigerte Leistungsaufnahme. Zusätzlich dazu müssen alternative Ausführungspfade oder unterschiedliche Daten zur Fehlererkennung genutzt werden, damit eine Erkennung von permanenten Fehler überhaupt gewährleistet ist. In der Literatur werden diese Techniken oft unter dem Begriff SIHFT (*software-implemented hardware fault tolerance*) zusammengefasst.

Verfahren der *offline*-Fehlererkennung beschränken sich auf das Ausführen eines Tests. Die Umsetzung eines Testverfahrens mit Hilfe eines ATE (*automatic test equipment*) ist nur direkt nach der Produktion möglich und eignet sich nicht als Vorstufe für Rekonfigurationen im Feld. Die Realisierung eines Tests im Feld lässt sich zum einen mit dem sogenannten BIST (*built-in self-test*) oder auch durch einen SBST (*software-based self-test*) umsetzen.

BIST basierte Verfahren erzeugen die notwendigen Testmuster entweder *on-chip*, mit Hilfe eines Testmustergenerators, oder halten sie in einem separaten Speicher vor. Das Spektrum der Möglichkeiten zur Erzeugung von Testmustern auf dem Chip reicht von Berechnungen durch zusätzliche Prozessoren bis hin zur Nutzung von Pseudozufallszahlen, erzeugt durch LFSR-Strukturen (*linear feedback shift register*). Nach der Erzeugung der Testmuster werden diese zur Umsetzung eines Scantests genutzt, dessen Antworten anschließend analysiert werden. Dazu werden die Antworten beispielsweise in MISR-Strukturen (*multiple input signature register*) zu Signaturen zusammengefasst, welche durch anschließenden Vergleich mit Referenzsignaturen eine Fehlererkennung leisten können [Ste00, WWW06].

Durch den Einsatz von BIST-Verfahren verringert sich die Testzeit nach der Produktion, da hier das ATE (*automatic test equipment*) keine Testmuster, sondern lediglich Steuerinformation für den BIST auf den Chip übertragen muss. Der Einsatz von BIST-Verfahren ist vor allem für Speicher schon lange etabliert, aber auch der Test von Prozessorstrukturen mit BIST-Verfahren ist weit verbreitet. Das Erstellen der BIST-Strukturen und der notwendigen Testmuster wird durch kommerzielle Programme weitgehend automatisch übernommen. Erweiterungen von BIST-Verfahren ermöglichen eine Nutzung nicht nur nach der Produktion, sondern auch im Feld [DAKP07]. Der BIST kann dann im Feld sowohl als *start-up* als auch als periodischer Test zur Fehlererkennung eingesetzt werden.

Fehlererkennung durch sogenannte SBST-Verfahren [TA80,APGP07,CMAB09] stellen eine Alternative zum eben vorgestellten BIST dar. Bei diesen Verfahren handelt es sich um einen funktionalen Test, bei dem eine Software auf dem zu testenden Prozessor ausgeführt wird. Durch den gezielten Entwurf von funktionalen Testmustern und dem Abbilden dieser auf Software entsteht ein strukturelles Testverfahren, welches in einem funktionalen Modus an die Schaltung appliziert wird. Somit stellt der SBST eine Alternative zum BIST dar, denn er kommt ohne zusätzliche Hardware aus und vermeidet den Effekt des *overtesting*[1]. Weiterhin sind SBST-Verfahren *at-speed* Tests, da die Schaltung während des Tests mit ihrem Arbeitstakt betrieben wird.

Die Fehlerüberdeckung, vor allem für Komponenten des Datenpfades, ist mit der von BIST-Verfahren vergleichbar [PGSR10, PZK07]. Zur Erstellung eines SBST gibt es funktionale aber auch strukturelle Methoden. Die Generierung von funktionalen Methoden basiert auf dem Befehlssatz der jeweiligen Architektur und kann rein zufällig oder durch Verfahren mit Rückkoppelung ablaufen.

In [PML02] wird ein zufälliges Verfahren vorgestellt. Hier erzeugt ein Kernel ausführbare Programme, die anschließend ausgeführt werden. VERTIS [SA98] erzeugt ebenfalls zufällige Instruktionssequenzen für die Komponenten eines Prozessors. Zusätzlich werden die jeweiligen Testantworten in Signaturen, erzeugt durch verfügbare Befehle, zusammengefasst.

In [CSRS04] wird eine Methode namens *micro GP* vorgestellt. Diese nutzt als Rückkopplung Informationen aus einer Fehlersimulation, um die Größe von zufällig erzeugten Instruktionssequenzen zu verringern und damit eine Steigerung der Effizienz des SBST zu erreichen. Strukturelle Verfahren zur Erzeugung eines SBST basieren auf Informationen aus System, RTL-Beschreibungen (*regis-*

[1] bezeichnet das Testen auf Fehler, welche im funktionalen Betrieb kein Fehlverhalten produzieren

ter transfer level) oder Gatternetzlisten. Der Vorteil einiger dieser Verfahren ist die Verfügbarkeit einer strukturellen Fehlerüberdeckung und das Verschieben der Testmustergenerierung für komplexe Logikfunktionen in ein ATPG-Programm (*automatic test pattern generation*). Hierdurch werden deterministische Testmuster erzeugt, welche dann in geeigneter Weise auf Sequenzen von Instruktionen abzubilden sind [CD00, CWLG07] [2, 12].

In [PGH+06] wird eine Möglichkeit beschrieben, einen SBST zu generieren, welcher die Pipeline eines Prozessors testet. Dieses Verfahren arbeitet auf einer sehr hohen Abstraktionsebene und nutzt lediglich einfache Parameter der Pipeline und des Speichersystems.

Ein kombinierter Test, welcher sowohl auf BIST und SBST zurückgreift, wurde für eine VLIW-Architektur vorgestellt [18]. In einer ersten Phase wird ein konventioneller Test mit Hilfe von Scanpfaden durchgeführt, um die korrekte Ausführung des anschließenden SBST zu gewährleisten. Durch einzelne Erweiterungen der Hardware dieser VLIW-Architektur konnte auf die Verwendung eines BIST verzichtet werden [16]. Hier werden einzelne diagnostische Testprogramme ausgeführt und erstellen in einem separaten Speicher ein Fehler-Syndrom. Anschließend wird das SBST durch einen zusätzlichen Prozessor an das jeweilige Fehler-Syndrom angepasst. Damit wird eine feingranulare Diagnose einzelner Komponenten mit einem SBST erreicht.

Viele der vorgestellten SBST-Verfahren sollen die Qualität des Produktionstests erhöhen. Allerdings ist die Nutzung der jeweils erstellten Testprogramm ohne größere Einschränkungen auch im Feld möglich. Damit können SBST-Verfahren im Feld durch einen *start-up* oder zyklischen Test eine Fehlererkennung leisten.

3.1.4.2 Fehlerdiagnose

Bevor die Rekonfiguration eines Systems stattfinden kann, muss der Fehlererkennung eine Fehlerdiagnose folgen. Der Prozess der Fehlerdiagnose wird immer dann notwendig, wenn das verwendete Verfahren zur Fehlererkennung diese nicht oder nur unzureichend leistet. Grundsätzlich muss eine der Rekonfiguration vorgeschaltete Diagnose eine Fehlerlokalisierung mindestens in der Granularität der Rekonfigurationsfunktion liefern. Einige der vorgestellten Verfahren zur Fehlererkennung liefern diese Lokalisierung zusätzlich zur Fehlererkennung und machen damit einen zusätzlichen Prozess der Fehlerdiagnose obsolet.

Die Verwendung einer zusätzlichen Fehlerdiagnose ist abhängig von den benötigten Informationen, die diese liefern soll. Dazu zählen beispielsweise Größe, Ort und der Zeitpunkt des Auftretens eines Fehlers [Ise06]. Für Rekonfigurationsstrategien ist vorrangig das genaue Lokalisieren des Fehlers notwendig.

Durch die Verwendung von Verzögerungssensoren an logischen Pfaden oder auch durch den Einsatz spezieller diagnostischer Schaltungen kann eine Lokalisierung des Fehlerorts erreicht werden. In [ABMF04] wird die Eingrenzung des Fehlerorts auf einzelne kombinatorische Blöcke durch die Verwendung eines zusätzlichen Schattenregisters erreicht. Dieses Register speichert die Werte der Kombinatorik mit einer zeitlichen Verzögerung zum eigentlichen Register. Durch eine solche Schaltung kann eine Verschlechterung der Schaltungsparameter entdeckt und eingegrenzt werden, bevor es zu einem permanenten Fehler kommt.

Die Autoren in [BFGM07] führen ein 3 stufiges Verfahren zur Diagnose einzelner Komponenten ein. Hier wird eine Vielzahl von unterschiedlich verzögerten Pfaden aus der jeweils observierten Komponente zu einer zusätzlichen Hardware-Komponente geleitet. Diese ist anschließend in der Lage, eine Verschlechterung der Schaltungsparameter in verschiedenen Abstufungen zu diagnostizieren. Damit wird eine entsprechende Anpassung des Arbeitstaktes möglich, was die Autoren hier zur Kalibrierung des Systems verwenden.

3.1.4.3 Rekonfiguration

Systeme, die nach einer Rekonfiguration keine Einschränkungen in der Verarbeitungsleistung aufweisen, besitzen zusätzliche Ersatzkomponenten, welche die Aufgaben defekter Komponenten vollständig übernehmen können. Führt die Rekonfiguration jedoch nur zu einer Isolation defekter Baugruppen, verschlechtert sich entweder die Verarbeitungsleistung oder die Funktionalität des Systems wird reduziert. Im zweiten Fall wird in der Literatur der Begriff Degradation verwendet. Eine Degradation findet immer dann statt, wenn es keine expliziten Ersatzkomponenten gibt. Auch im Falle einer erschöpften expliziten Redundanz kann es zur Degradation des Systems kommen. Weiterhin können beide Arten von Rekonfiguration nochmals unterteilt werden. Es existieren Strategien, welche direkt nach der Fertigung als auch im Feld Anwendung finden. Rekonfiguration nach der Fertigung wird zur Steigerung der Produktionsausbeute verwendet und soll den Effekt der statischen Variation verringern. Durch diese Strategien müssen Schaltungen mit Produktionsfehlern nicht zwingend als Ausschuss betrachtet werden. Vielmehr entstehen unterschiedliche Qualitätsstufen für dieselbe Schaltung. Somit wird es möglich auch Schaltungen zur veräußern, die prinzipiell die

geforderte Spezifikation nicht erfüllen. Rekonfigurationsstrategien im Feld sorgen für eine Verlängerung der Lebensdauer beziehungsweise einer Steigerung der Zuverlässigkeit. Diese Techniken sorgen für einen fortlaufenden korrekten Betrieb des Systems trotz auftretender Fehler durch dynamische Variation oder zufällige Fehler. Auftretende Fehler führen dann nicht zwangsweise zum Ausfall des Gesamtsystems, auch wenn die eigentliche Spezifikation des Systems nicht mehr erfüllt ist. Oft lassen sich Verfahren zur Rekonfiguration im Feld auch für die Kompensation von Fehlern nach der Produktion einsetzen. In Abbildung 3.6 findet sich die eben beschrieben Unterteilung von Verfahren der Rekonfiguration.

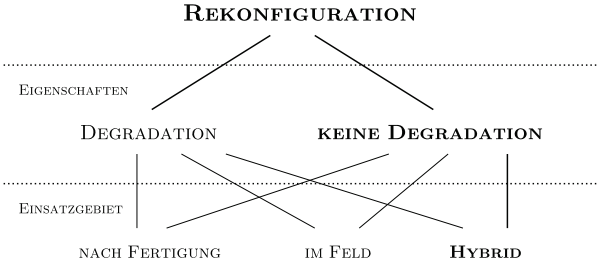


Abbildung 3.6: Unterteilung möglicher Strategien der Rekonfiguration

Rekonfiguration für Speicher

Rekonfigurationsstrategien zur Kompensation von Fehlern in hoch regulären Strukturen wie Speichern [BSOS04, KZK+98, PASM08, TWH+07, HCW06, LY-HW03] aber auch in FPGAs (*field programmable gate array*) [MHS+04, HKV06] sind längst Stand der Technik. Der Einsatz solcher Strategien für reguläre Strukturen lässt sich vor allem durch den geringen zusätzlichen Aufwand an Hardware erklären. Zusätzlich dazu besteht ein großer Anteil heutiger SoCs aus eben diesen Strukturen [MPKSZ05]. Demzufolge ist die Absicherung von Caches und Speichern durch Redundanz und Rekonfiguration ein sehr effektives Mittel, um auf Fehler in SoCs zu reagieren.

Abbildung 3.7 zeigt den grundsätzlichen Aufbau einer solchen Rekonfiguration für Speicherbausteine. Zusätzlich zum eigentlichen Speicherfeld existieren redundante Zeilen und/oder Spalten, welche im Fehlerfall die Aufgabe defekter Zeilen und/oder Spalten übernehmen. Durch konfigurierbare Dekoder wird die Nutzung der redundanten anstelle der defekten Speicherzellen realisiert. Die Rekonfiguration des Dekoders kann direkt nach der Produktion durch das Durchbrennen von eingebauten Sicherungen vorgenommen werden und dient dann vorrangig zur

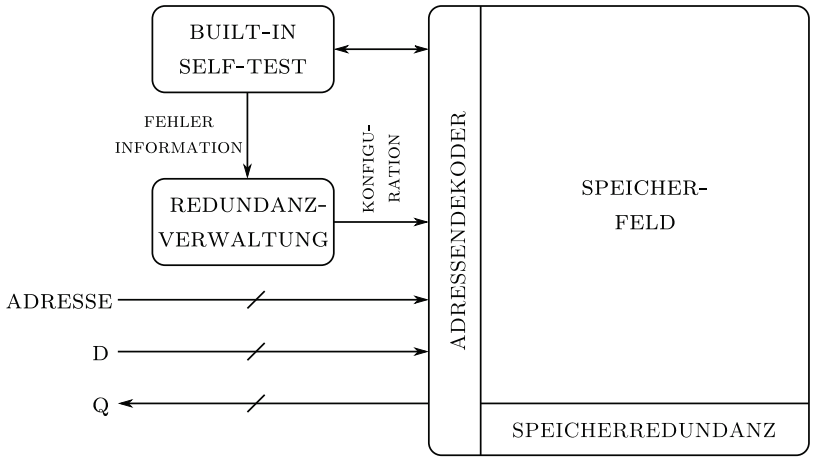


Abbildung 3.7: Schematischer Aufbau eines Speichers mit eingebauter Rekonfigurationsfunktion

Steigerung der Produktionsaubeute. Hierzu sind in der Regel hohe Ströme notwendig, die im Feld nicht zur Verfügung stehen. Durch die Entwicklung spezieller E-Fuses [ACF+03, SPP01] konnte der Prozess der Rekonfiguration aus der Produktion ins Feld verschoben werden. Somit wurde der Einsatz dieser Verfahren auch zur Kompensation von permanenten Fehlern, beispielsweise durch Frühausfälle oder Alterungseffekte, möglich.

Eine weitere Möglichkeit zur Steigerung der Ausbeute wird durch die Autoren in [APR04] vorgestellt. Die Anwendung dieses Verfahrens ist ausschließlich für Cache-Strukturen möglich. Fehler werden ebenfalls durch einen BIST erkannt und an einen Controller übermittelt. Dieser reduziert dann durch Konfiguration der Dekoder die mögliche nutzbare Größe des Speichers. Durch die Reduzierung der Speichergröße entsteht hier eine Verringerung der Verarbeitungsleistung.

In [SYW05] wurde eine Kombination aus Fehlererkennung durch ECC und anschließender Rekonfiguration umgesetzt. Auftretende Fehler werden durch die Verwendung von ECC kompensiert. Anschließend wird in Leerlaufphasen eine Rekonfiguration durchgeführt, welche die Integrität der Schaltung wiederherstellt.

Für FPGAs existieren Verfahren, die mehrere bereits im Vorfeld kompilierte Konfigurationen bereitstellen. Im Fehlerfall kann eine alternative Konfiguration verwendet werden, welche somit eine Kompensation von permanenten Fehlern leistet [HM01].

Rekonfiguration für Logik

Eine Steigerung der Ausbeute für Logik kann durch einfache Verdopplung von Transistoren oder Gattern erreicht werden [SPR04, KVC+06]. Auch der Einsatz von Ersatzkomponenten für größere Komponenten wurde bereits publiziert [BBSS98]. Die Verwendung der Ersatzkomponenten wird nach der Produktion durch ein Durchbrennen von Sicherungen realisiert.

Mit dem Projekt HYETI (*high yield and error tolerant integration*) [LKK+94] wurde bereits ein Mikroprozessor vorgestellt, welcher durch verschiedene Verfahren eine Toleranz gegenüber Fertigungsfehlern erreicht. Dazu werden die irregulären Strukturen des Kontrollpfades in ein PLA (*programmable logic array*) abgebildet. Die PLA Struktur kann dann mit den bereits vorgestellten Verfahren der Speicherrekonfiguration gegenüber permanenten Fehlern abgesichert werden. Für die regulären Strukturen des Datenpfades wurden Redundanzen implementiert. Produktionsfehler werden hier durch einen Scantest festgestellt. Das Durchbrennen von Sicherungen mit Hilfe eines Lasers oder auch die Programmierung von *floating-gates*[2] wird anschließend zum dauerhaften Einstellen der Konfiguration verwendet.

Die Autoren in [SKMB03] implementieren unterschiedliche Redundanzstrategien, um auf Produktionsfehler reagieren zu können. Die verwendeten Redundanzen hier sind CLR (*component level redundancy*), AR (*array redundancy*) und DQR (*dynamic queue redundancy*). In CLR werden Komponenten repliziert und bieten eine zusätzliche Steigerung der Leistung durch die dann vorhandene Parallelität. Allerdings ist die korrekte Funktionalität der Schaltung auch mit weniger Komponenten gegeben. Durch den eingebauten Selbsttest werden fehlerhafte Komponenten identifiziert und dauerhaft als belegt gekennzeichnet. Somit wird die Nutzung fehlerhafter Ressourcen vermieden, was zu einer Verringerung der Verarbeitungsleistung durch den Wegfall von Parallelität führt. Der AR Mechanismus implementiert redundante Zeilen in Speicherfeldern. Im Falle eines Defekts im Speicherfeld lassen sich die Dekodierer zur Adressierung so konfigurieren, dass die redundanten anstelle der defekten Zeilen verwendet werden. Dieses Verfahren verursacht keine Leistungsverringerung. Bei DQR werden Puffer-Strukturen durch ein zusätzliches Bit versehen. Im Fehlerfall wird dieses Bit permanent ausgeschaltet, dies führt bei der Abarbeitung der Warteschlange zu zusätzlichen Verzögerungen, also zusätzlicher Abarbeitungszeit, und damit einer Leistungsverringerung.

[2] dt. nicht angeschlossene Steuerelektrode - ein spezieller Transistor der zum permanenten Speichern verwendet wird

In [RK11] verwenden die Autoren eine zusätzliche Integer-ALU, welche als Ersatzkomponente in einem Mehrkernprozessor zum Einsatz kommt. Durch implementierte Multiplexer-Strukturen ist ein Austausch defekter ALU in beliebigen Kernen möglich. Falls mehrere ALUs ausfallen findet eine zeitliche Zuteilung der Ersatzkomponente statt. Eine Degradation entsteht durch den Ausfall von mindestens 2 ALUs und entspricht einer verlängerten Abarbeitungszeit.

Ein ganz ähnliches Prinzip verfolgen die Autoren in [PKK09] für FPUs in Mehrkernprozessoren. Hier wird jedoch keine zusätzliche FPU benötigt. Unter der Annahme von wenig ausgelasteten FPUs in Mehrkernprozessoren wird ein Mechanismus vorgestellt, der Berechnungen auf defekten FPUs in andere Kerne verschiebt und Ergebnisse wieder zurückleitet. Durch das Umleiten der Berechnungsaufgaben und die Möglichkeit bereits belegter FPUs führt auch diese Methode zur Verzögerung der Abarbeitungszeit.

Das eben vorgestellte Prinzip wurde durch die Autoren in [RS08] für alle 5 Phasen einer Pipeline in einem Mehrkernprozessor realisiert. Ist eine bestimmte Phase der Pipeline in einem Prozessor fehlerhaft, werden ihre Eingänge an die gleiche Phase eines anderen Prozessor geleitet. Diese übernimmt dann die Aufgabe der defekten Phase und leitet die Ergebnisse zurück. Mit dem Auftreten des ersten Fehlers im Mehrkernprozessor kommt es zu einer möglichen Verzögerung bei der Abarbeitung, da nicht mehr alle Prozessoren gleichzeitig arbeiten können.

Die Autoren in [SCP+06] implementieren zusätzliche BIST-Strukturen für die einzelnen Komponenten einer Pipeline eines VLIW-Prozessors. Durch einen zyklischen Test der einzelnen Komponenten werden Fehler erkannt. Fehlerhafte Baugruppen werden im weiteren Betrieb nicht mehr verwendet. Die vorhandene Redundanz durch die gewählte VLIW-Architektur ermöglicht dann einen degradierten Betrieb des Prozessors im Fehlerfall. In [Sch10] wird ebenfalls die vorhandene Redundanz einer VLIW-Architektur zur Kompensation permanenter Fehler genutzt. Dieses Verfahren stellt eine grobgranulare Rekonfiguration dar. Zusätzliche Strukturen ermöglichen eine Anpassung der Software auf bestimmte Fehlersituationen und verschieben dafür Instruktionen von fehlerhaften Ausführungspfaden auf fehlerfreie. Erweiterungen dieses Ansatzes zur fein granularen Kompensation von permanenten Fehlern finden sich in [Sch11] und [16]. Grundsätzlich erfolgt auch hier eine Anpassung des statischen Ablaufplans auf bestimmte Fehlersituationen. Diese berücksichtigt allerdings detailliertere Informationen des Fehlerortes. Beispielsweise führt ein nicht lesbares Register eines Ausführungspfades nicht zur vollständigen Umplanung aller Befehle dieses Pfades, sondern lediglich zur Umplanung von Befehlen, die das nicht lesbare Register adressieren.

Mit DIVA (*dynamic implementation and verification architecture*) [Aus99] stellen die Autoren ein Verfahren vor, welches auf der Verwendung von zusätzlichen Checker-Komponenten basiert. Diese überprüfen die jeweilige Berechnung durch eine zusätzliche Komponente bevor das Ergebnis durch den Prozessor weiterverwendet werden kann. In [WA01] wird dieser Ansatz für den Alpha 21264 Mikroprozessor umgesetzt. Der zusätzliche Aufwand beträgt etwa 6% zusätzlicher Fläche und ca. 1.5% höherer Leistungsaufnahme. Zusätzliche Erweiterungen zu DIVA beschreiben die Autoren in [BSO05]. Hier werden wie in [SKMB03] für Pufferstrukturen zusätzliche Ersatzkomponenten implementiert. Somit wird es auch in DIVA möglich, das System für eine bestimmte Anzahl permanenter Fehler mit seiner ursprünglichen Verarbeitungsleistung weiterzubetreiben.

3.1.4.4 Fehlerkorrektur

Nachdem eine Rekonfiguration des Systems defekte Komponenten ausgetauscht oder abgeschaltet hat, kann je nach Anforderung noch eine Korrektur des Fehlers folgen. Wurde der Rekonfigurationsprozess durch eine *online*-Fehlererkennung gestartet, ist in der Regel eine einfache Wiederholung der fehlerhaften Ausführung als Korrekturmaßnahme ausreichend.

War das auslösende Ereignis der Rekonfiguration eine *offline*-Fehlererkennung und ist laut Anforderungen keine Korrektur notwendig, dann kann der reguläre Betrieb des Systems durch einen einfachen Neustart eingeleitet werden. Ist hingegen eine Korrektur zwingend erforderlich, wird in der Literatur vor allem eine Umsetzung mit Hilfe von Checkpoints vorgeschlagen [SCP+06,DQ11,KK07]. Hierbei wird in zyklischen Intervallen ein Punkt in der Programmausführung erstellt, zu dem im Falle eines auftretenden Fehlers ein Rücksprung erfolgen kann. Der Einsatz von Checkpoints und Rücksprüngen ist vor allem in höheren Ebenen der Datenverarbeitung angesiedelt.

3.2 Variation aware design

Im Gegensatz zur Fehlertoleranz, welche lediglich auf Fehler reagieren kann, werden in diesem Abschnitt Verfahren vorgestellt, welche das Auftreten von Fehlern verzögern, abschwächen oder ganz verhindern können. Zur Umsetzung solcher Maßnahmen muss die genaue Fehlerursache in der Regel bekannt sein.

Kleinere Fertigungsgrößen und sinkende Versorgungsspannungen sorgen immer mehr dafür, dass die Effekte der statischen und dynamischen Variation an Relevanz zunehmen und eine fehlerfreie Produktion oder einen zuverlässigen Betrieb erschweren. Dennoch werden zufällige Fehlerursachen auch für kleinere Fertigungsgrößen nicht weniger relevant. Der Einsatz von Techniken, welche sich einer stärkeren Variation bewusst sind, zielt sowohl auf den Produktionsprozess als auch auf einzelne Schritte des Entwurfsprozesses. Folgend werden Möglichkeiten des *variation aware design* für den Entwurfsprozess vorgestellt. Möglichkeiten zur Abschwächung der Effekte im Produktionsprozess beziehen sich auf Arbeiten in der Halbleiterphysik und werden an dieser Stelle nicht weiter betrachtet.

3.2.1 Entwurfsprozess

Während des Entwurfsprozesses von Systemen kann ebenso gezielt auf mögliche Variationen in Produktion und Betrieb von integrierten Schaltungen reagiert werden. Sind beispielsweise bestimmte produktionsbedingte oder zeitliche Variationen im Schaltverhalten der Transistoren bekannt, dann kann der Entwurfsprozess für Schaltungen diese Variationen berücksichtigen.

Eine Ansatz an dieser Stelle ist die Einführung von *guard bands* [Ala08]. Hier werden härtere Bedingungen an den Entwurf von Schaltungen gestellt als eigentlich benötigt. Diese härteren Bedingungen ermöglichen einen zusätzlichen Zeitpuffer, welcher dann während einer zeitlichen Variation allmählich aufgebraucht wird. Solange die Variation sich in diesem Pufferbereich befindet, hat sie keine Auswirkungen auf den korrekten Betrieb des Systems. Beispielsweise können Schaltungen für den Betrieb bei einer bestimmten Frequenz entworfen, jedoch im Einsatz mit einer niedrigeren betrieben werden.

In [KKS07] wird ein Methode vorgestellt, welche während der Synthese eine Verbesserung der Schaltungsintegrität erreicht. Dazu findet eine zeitliche Charakterisierung aller Gatter einer Bibliothek für den Effekt NBTI statt. Anschließend fließen diese Informationen in den Syntheseprozess ein und verringern NBTI für

besonders gefährdete Gatter.

Zur Verringerung des Effekts der Elektromigration stellen die Autoren in [LJ05] einen optimierten Layoutprozess vor. Hier wird während des Prozesses der Platzierung und Verdrahtung auf hohe Stromdichten geachtet. Werden bestimmte Grenzwerte erreicht, findet eine Anpassung der betroffenen Strukturen statt.

Ein weiterer Ansatz [AVU+08] zur Verringerung von Elektromigration setzt auf eine Umkehrung der Stromrichtung. Dazu wird eine zusätzliche Steuerung in die Schaltung integriert, welche in der Lage ist, die Versorgungsspannung mit der Masse zu vertauschen. Durch diese Umkehr kann die Ausbildung von Störstellen durch Elektromigration teilweise wieder rückgängig gemacht werden.

Die gezielte Reduzierung von Stress in Schaltungen taucht als weitere Herangehensweise immer wieder in der Literatur auf [Ala08, ABMC08].

Für Schaltungen ohne Redundanz kommen beispielsweise Verfahren zum Einsatz, welche mit Hilfe von Thermalmanagement eine gleichmäßige Verteilung der Belastungen auf die einzelnen Ressourcen einstellt. Diese Möglichkeit eröffnet sich vor allem bei heutigen Mehrkernsystemen.

Die Autoren in [MLN+06] stellen ein compilergestütztes Thermalmanagement zur Lastverteilung der funktionalen Einheiten eines VLIW-Prozessors vor. Da in diesen Architekturen eine statische Ablaufplanung existiert, kann schon bei der Übersetzungszeit gezielt auf eine einseitige Belastung bestimmter Baugruppen reagiert werden.

Wurde das System mit Redundanz zur Kompensation von permanenten Fehlern ausgestattet, dann bietet sich eine Verteilung der Belastungen auf redundante Komponenten an. Die Autoren in [SZBP08] beschreiben ein Verfahren, welches durch einen zyklischen Austausch durch Ersatzkomponenten in SRAM Bausteinen eine Verlängerung der Lebensdauer erreicht.

Ein ganz ähnliches Verfahren wurde in [17] für die funktionalen Einheiten einer VLIW-Architektur beschrieben. Hier geht es um eine Reduzierung der Temperaturen durch den zyklischen Austausch aller funktionalen Einheiten einschließlich ihrer Ersatzkomponenten.

3.3 Zusammenfassung

In diesem Kapitel wurden Möglichkeiten der Fehlertoleranz als auch des *variation aware design* vorgestellt. Sind die Fehlerursachen im Detail geklärt, folgen Anpassungen von Produktions- und/oder Entwurfsprozess, welche dazu führen können, dass der Einsatz von Fehlertoleranz unnötig wird. Für Fehlereffekte, deren Ursache unzureichend geklärt ist oder für die keine Lösungen zur Behebung bekannt sind, bleibt der Einsatz von Fehlertoleranz unvermeidlich. Der Abschnitt zur Fehlertoleranz bezog sich im Wesentlichen auf Maßnahmen der Rekonfiguration, welche eine Kompensation von permanenten Fehlern leistet. Die steigende Anfälligkeit von Schaltungen durch sinkende Strukturgrößen macht es notwendig, Techniken zur Kompensation permanenter Fehler auch für Prozessoren weiterzuentwickeln, welche sowohl nach der Produktion als auch im Feld zum Einsatz kommen können. Gleichzeitig bleibt die Notwendigkeit zur Kompensation von transienten und zunehmend auch intermittierenden Fehlern bestehen.

Trotz unterschiedlichster Lösungen zur Kompensation von permanenten Fehlern in integrierten Schaltungen ist dem Autor kein Verfahren bekannt, welches sich mit der effizienten Auswahl von Hardware-Redundanz für Prozessorkomponenten beschäftigt. Gerade durch die Entwicklung in der Halbleiterindustrie und den damit verbundenen Effekten geringerer Zuverlässigkeit, sinkender Lebensdauer und Verschlechterung der Produktionsausbeute integrierter Schaltungen ist die effiziente zielgerichtete Auswahl von Hardware-Redundanz notwendig.

KAPITEL 4

Rekonfiguration für Prozessorkomponenten

Der erste Teil dieses Kapitels stellt eine skalierbare Architektur vor, welche zuschaltbare Redundanz für Prozessorkomponenten bietet. Dabei wird im Detail auf den Aufbau, die Möglichkeiten der Fehlerisolation und die Umsetzung der Administration der Redundanz eingegangen. Anschließend werden Möglichkeiten der Umsetzung von Fehlererkennung und Diagnose für permanente Fehler im Feld vorgestellt und diskutiert. Im dritten Teil dieses Kapitels folgt die Erläuterung der Funktionsweise des Gesamtsystems in ihren jeweiligen Anwendungsszenarien. Abschließend wird gezeigt, wie die vorgestellte Architektur und der zielorientierte Auswahlprozess in einen bestehenden Entwurfsprozess zu integrieren sind.

4.1 Skalierbare Hardware-Redundanz

Der Aufbau der Hardware-Redundanz zum Zweck der Kompensation permanenter Fehler in dieser Arbeit orientiert sich an folgenden Anforderungen:

- Dauerhaftes Fehlverhalten durch permanente Fehler soll durch Rekonfiguration für den weiteren Betrieb vermieden werden. Die Korrektur der Auswirkungen von Fehlern die sich bereits im System manifestiert haben ist nicht Teil des Konzeptes und ist, falls notwendig, durch zusätzliche Maßnahmen der Fehlermaskierung bzw. Fehlererkennung und Korrektur zu behandeln. Dafür soll der Ansatz so gestaltet sein, dass eine Erweiterung beziehungsweise eine Integration dieser zusätzlichen Maßnahmen möglich bleibt.

- Die Ursache des permanenten Fehlers spielt für diesen Ansatz keine Rolle. Es kann sich dabei um Fertigungsfehler, Frühausfälle, zufällige Fehler oder auch Alterungsfehler handeln. Aus diesem Grund muß die Architektur alle notwendigen Komponenten zur Rekonfiguration auf dem Chip integriert haben und der Ansatz muß ebenso im Feld anwendbar sein.

- Der Aufbau muss verschiedene Konfigurationen ermöglichen, um einen zielgerichteten Auswahlprozess der Redundanz zu erlauben.

- Der Ansatz soll ohne zusätzliche Veränderungen der zugrunde liegenden Synthesebibliothek auskommen. Damit wird zum einen die Anwendbarkeit und zum anderen die Möglichkeit der Integration in bestehende Entwurfsprozesse gewährleistet.

- Durch eine Rekonfiguration des Systems soll keine Degradierung des Systems stattfinden, weil die Ressourcen heutiger Speziallösungen bereits optimiert sind und daher ihre Funktion nach einer Degradierung nicht mehr erfüllen können. Daher müssen die verfügbaren Ressourcen vor und nach der Rekonfiguration identisch sein.

- Optional soll dieser Ansatz eine Lastverteilung integrierter redundanter Ressourcen ermöglichen. Damit würde die Rekonfiguration dieser Arbeit weitere Optimierungsmöglichkeiten bezüglich der Lebenszeitsteigerung bereitstellen.

4.1.1 Redundante Logik-Blöcke (RLB)

Dieser Abschnitt beinhaltet den grundsätzlichen Aufbau der Hardware-Redundanz, welche essentieller Bestandteil der Kompensation permanenter Fehler darstellt. Das wesentliche Prinzip ist in Abbildung 4.1 zu sehen und orientiert sich an bereits bestehenden Konzepten zur Kompensation von permanenten Fehlern in Speicherbausteinen.

Hier sind für eine Menge M identischer Logik-Funktionen, dargestellt als Logik-Blöcke (LB), eine zweite Menge K identischer Ersatz-Logik-Blöcke (ELB) definiert. Die Menge K der Ersatz-Logik-Blöcke bestimmt gleichzeitig die Anzahl der fehlerhaften Logik-Blöcke die mit dieser Implementierung ausgetauscht werden können. Zusätzlich befinden sich vor den jeweiligen Ersatz-Logik-Blöcken und hinter den eigentlichen Logik-Blöcken Multiplexer, die ein beliebiges Umleiten jeder Funktion auf jeden der redundanten Blöcke erlauben. Die Ansteuerung

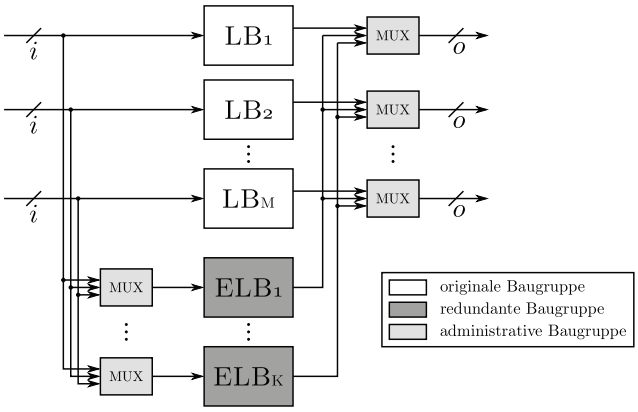


Abbildung 4.1: Redundanter Logik-Block

der Multiplexer und damit die Administration der Redundanz wird in späteren Abschnitten genauer erläutert und wird aus diesem Grund hier nicht näher betrachtet.

Die Größe der LB ist prinzipiell frei wählbar und es ist ebenfalls nicht ausgeschlossen, dass Logik-Blöcke auch speichernde Elemente wie einzelne Register oder ganz Speicherbänke beinhalten. Sequentielle Schaltungen als LB bringen lediglich für den Test zusätzliche Einschränkungen mit sich.

Die Umsetzung der redundanten Logik-Blöcke aus Abbildung 4.1 ist skalierbar für beliebige M und K. Damit kann eine Implementierung der RLB in Abhängigkeit der zu Verfügung stehenden Komponenten in einem speziellen Design erfolgen. Die Anzahl der Ersatz-Logik-Blöcke K ist für diese Art der Implementierung beliebig wählbar. Die exakte Anzahl wird in dieser Arbeit durch einen analytischen Entscheidungsprozess festgelegt. Dieser berücksichtigt bestimmte Ziele wie zu erreichende Zuverlässigkeitssteigerung, Lebensdauer oder auch Produktionsausbeute bei einem maximal zulässigen Hardware-Mehraufwand und wird im nächsten Kapitel im Detail erläutert.

Weiterhin ist die Größe der Logik-Blöcke beliebig wählbar und kann einzelne Gatter aber auch komplexe Prozessorkomponenten beinhalten. Jedoch ist der zusätzliche Aufwand an Hardware für kleine Komponenten wie Gatter deutlich höher als beispielsweise für ALUs. Wichtig ist hier allerdings, dass nicht nur die Größe der Logik-Blöcke sondern auch weitere Eigenschaften wie deren Anzahl und deren Menge von Ein- und Ausgabesignalen den zusätzlichen Hardware-Mehraufwand und damit auch die Effizienz beeinflussen. Dies macht eine detaillierte Betrachtung aller möglichen Funktionen als Logik-Block notwendig. Voran-

gegangene Arbeiten haben jedoch gezeigt, dass eine Untersuchung von Strukturen unter 400 Transistoren in der Regel einen derart hohen Hardware-Mehraufwand bedeuten, dass kein positiver Effekt zu verzeichnen ist, wenn sie durch die vorgeschlagene Ersetzungsfunktion erweitert werden [3].

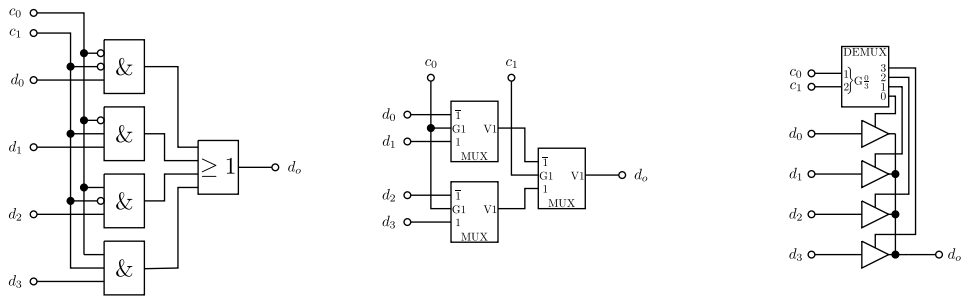
4.1.2 Schaltnetzwerke

Die vorgeschlagene Implementierung von RLB aus dem vorherigen Abschnitt 4.1.1 benötigt zusätzlich zu den Ersatz-Logik-Blöcken (ELB) auch Multiplexer Strukturen, die das Umleiten von Signalen ermöglichen um die Rekonfiguration der Hardware zu realisieren. Diese sind im Abschnitt 4.1.1 als Multiplexer bezeichnet, was ihrer grundsätzlichen Funktion entspricht. Die Umsetzung eines Multiplexers in CMOS kann durch unterschiedliche Schaltungen von Transistoren erreicht werden. Die Entscheidung für eine spezifische Umsetzung, beispielsweise *pass transistor logic* (PTL), *double pass transistor logic* (DPL), *transmission gate* oder *pull-down* NMOS und *pull-up* PMOS Netzwerke, trifft der Halbleiterhersteller in Abhängigkeit des Herstellungsprozesses und den jeweiligen Anforderungen an Größe, Schaltverhalten und Stromverbrauch. Gerade beim *semi-custom* Entwurf hat der Entwickler der Schaltung auf die jeweilige Umsetzung der einzelnen Elemente einer gegebenen Zellbibliothek keine Einfluss mehr.

Die Beschreibung eines Multiplexers auf RT-Ebene wird durch das Syntheseprogramm auf vorhandenen Zellen der Bibliothek abgebildet. Dadurch kann eine Schaltung mit verschiedenen Gattertypen entstehen (siehe Abbildung 4.4(a)). Die entstehende Multiplexerschaltung basiert dann auf Gattertypen der jeweiligen Zellbibliothek. Oft beinhaltet die Zellbibliothek explizite 2 zu 1 Multiplexer-Makros, welche dann durch das Syntheseprogramm genutzt werden (siehe Abbildung 4.4(b)). Eine dritte Möglichkeit ist der Aufbau eines Multiplexers durch Tri-State Gatter (Abbildung 4.2(c)). Diese Umsetzung muss der Entwickler der RT-Beschreibung durch explizite Verwendung von hochohmigen Signalpegeln erzwingen.

In dieser Arbeit wurde die RT-Beschreibung der RLB so implementiert, dass sämtliche Multiplexer durch Tri-State Gatter umgesetzt werden. Die Gründe für diese Umsetzung lassen sich wie folgt zusammenfassen:

- Tri-State Gatter sind nicht nur in der verwendeten Synthesebibliothek vorhanden, sondern sind für gewöhnlich fester Bestandteil jeder Zellbibliothek.



(a) Multiplexer aus Grundgattern

(b) Multiplexer-Makro

(c) Multiplexer aus Tri-State Gattern

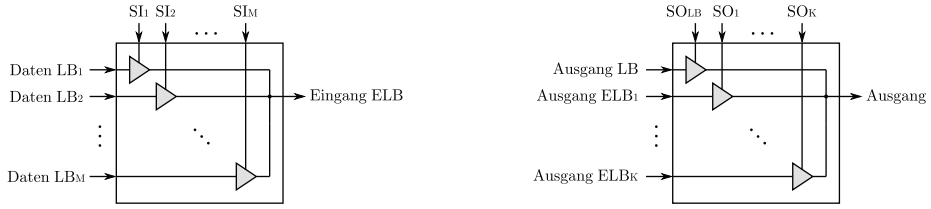
Abbildung 4.2: Typische Umsetzungen von 4 zu 1 Multiplexer mit den Möglichkeiten von Standard Zellbibliotheken

- Durch einen möglichen hochohmigen Ausgang können mehrere Tri-State Gatter zu unterschiedlichen Zeiten eine Signalleitung treiben. Damit ergeben sich typischerweise geringere Signalverzögerungungen als bei einer klassischen Multiplexer-Baumstruktur für große n bei n zu 1 Multiplexern [HH89]
- Der mögliche hochohmige Ausgang ermöglicht eine Fehlerisolation (siehe Abschnitt 4.1.3) durch die Abkoppelung der Signalpegel von fehlerhaften Schaltungsteilen.

Abbildung 4.3 zeigt den Aufbau der Multiplexer vor einem Ersatz-Logik-Block 4.3(a) und am Ausgang eines Logik-Blocks 4.3(b). Alle Steuersignale SI beziehungsweise SO eines Multiplexers werden anschließend durch separate Speicherzellen betrieben und benötigen daher keine Dekodierung.

4.1.3 Fehlerisolation der RLB

Die vorgestellten RLB aus Abbildung 4.1 sind in der Lage, jede Funktion der LB an einen beliebigen ELB zu leiten. Die Anzahl der Tri-State Gatter dieser Implementierung reicht aus, um die Funktion der Rekonfiguration zu gewährleisten. Problematisch an dieser Implementierung ist zum einen, dass keine Isolation defekter Logik-Blöcke umgesetzt wird. Damit steigt die dynamische Verlustleistung der Schaltung nach einer Rekonfiguration, welches zu ungewolltem Systemverhalten führen kann. Zum anderen sind die Eingänge der Ersatz-Logik-Blöcke



(a) Multiplexer am Eingang eines ELB (b) Multiplexer am Ausgang eines LB

Abbildung 4.3: Implementierungen der Multiplexer des RLB aus Abbildung 4.1

hochohmig, wenn sie nicht die Aufgabe eines Logik-Blocks erfüllen. In einer Simulation hat das keine Auswirkungen. Auf einem gefertigten Chip nehmen diese hochohmigen Eingänge jedoch dennoch Signalpegel an, was dann zu einem ungewollten Schaltverhalten der ELB führt. Aus diesem Grund sind die Eingänge von ungenutzten LB und ELB von den Signalpegeln zu trennen. Zusätzlich sollten die ungenutzten Blöcke keine hochohmigen Eingänge haben. In Abbildung 4.4 sind die Erweiterungen zur Abkoppelung von Logik- und Ersatz-Logik-Blöcken dargestellt.

(a) Multiplexer am Eingang der ELB (b) Multiplexer am Eingang der LB

Abbildung 4.4: Erweiterte Implementierungen der Multiplexer an den Eingängen von ELB und LB

Diese vorgestellte Erweiterung der Multiplexer aus Abbildung 4.4 ermöglicht zwar eine vollständige Isolation der Signale aller Logik- und Ersatz-Logik-Blöcke und damit auch eine vollständige Reduzierung der dynamischen Verlustleistung, damit ist aber noch nicht sichergestellt, dass fehlerhafte Komponenten keine ne-

gativen Auswirkungen auf andere Systemeigenschaften haben. Beispielsweise können durch Fehlereffekte entstandene Kurzschlüsse zu hohen Belastungen der Versorgungsspannung führen, diese Spannungsabfälle belasten dann unter Umständen nicht nur die betroffene Funktionseinheit sondern auch umliegende Baugruppen oder sogar den gesamten Chip. Aus diesem Grund ist eine Abkoppelung der fehlerhaften Blöcke von der Versorgungsspannung durchaus sinnvoll. Ein möglicher Aufbau ist in Abbildung 4.5 dargestellt.

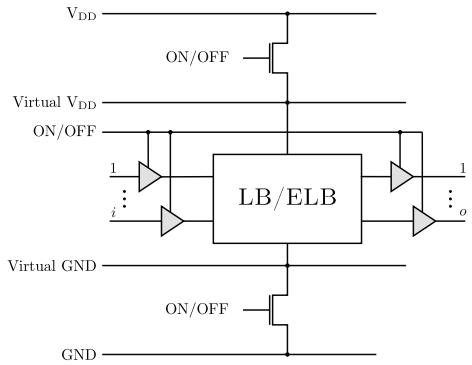


Abbildung 4.5: *Power gating* Schema für einen LB/ELB und Tri-State Gatter zur Isolation der Signalpegel

Die Steuerung der Abkoppelung betroffener Komponenten von der Versorgungsspannung kann durch verschiedene Implementierungen umgesetzt werden. Dazu stehen umkehrbare aber auch unumkehrbare Techniken zur Verfügung. Möglichkeiten der umkehrbaren Isolation von Teilschaltungen eines Designs sind aus dem Bereich des *low power* Designs bekannt. Dabei soll vor allem die statische Verlustleistung von nicht verwendeten Schaltungsteilen reduziert werden. Vorgeschlagen wurde das sogenannte *power gating* bereits für Caches [PYF+00] aber auch für Ausführungseinheiten [HBS+04].

Möglichkeiten zur unumkehrbaren Isolation von Schaltungsteilen werden vor allem nach der Produktion zur Reparatur von defekten Schaltungen verwendet. Der Einbau spezieller Metallleitungen, die nach der Produktion mit Hilfe eines Lasers durchtrennt werden können, ermöglicht es defekte Schaltungsteile elektrisch von der restlichen Schaltung zu trennen. Sinkende Strukturgrößen, der hohe Flächenaufwand für diese *laser fuses* und der hohe Aufwand beim Ausrichten des Lasers auf diese Leitungen machen diese Technik zunehmend unwirtschaftlich [ACF+03]. Alternativ können an Stelle von *laser fuses* auch *e-fuses* verwendet werden. Diese lassen sich durch zusätzliche Leitungen auf dem Chip ansteuern und benötigen keinen Laser [ACF+03].

In dieser Arbeit werden ausschließlich die Signalleitungen von nicht benutzten Blöcken isoliert. Eine Isolation der Versorgungsspannung kann durch zusätzliche Anweisungen bei Schaltungssynthese und Layout realisiert werden, wenn die verwendeten Bibliotheken dafür ausgelegt sind.

Die Verwendung von *power gating* kann an dieser Stelle eingesetzt werden, um die notwendige Isolation defekter Schaltungsteile zu realisieren. Diese Technik ist *on-chip* verfügbar und kann dadurch auch im Feld, zum Zwecke der Rekonfiguration, genutzt werden. Die Ansteuerung dieser *power gates* kann durch dieselbe Funktion realisiert werden wie die der Tri-State Gatter.

Im konkreten Einsatzfall sind folgende Effekte zusätzlich zu betrachten:

- Die Auswirkungen auf die Schaltung durch einen Abfall der Versorgungsspannung beim Ein-/Ausschalten bestimmter Schaltungsteile.

- Der Einfluss zusätzlicher Verzögerungen beim Zuschalten bestimmter Baugruppen. (Einschaltverzögerungen)

In Abbildung 4.6 ist die Implementierung redundanter Logik-Blöcke dieser Arbeit dargestellt. Die Multiplexer-Strukturen bestehen aus Tri-State Netzwerken und eine Isolation nicht verwendeter (Ersatz-)Logik-Blöcke wird durch weitere Tri-State Gatter erreicht, welche die Eingangspegel des jeweiligen Blocks auf logisch 0 einstellen.

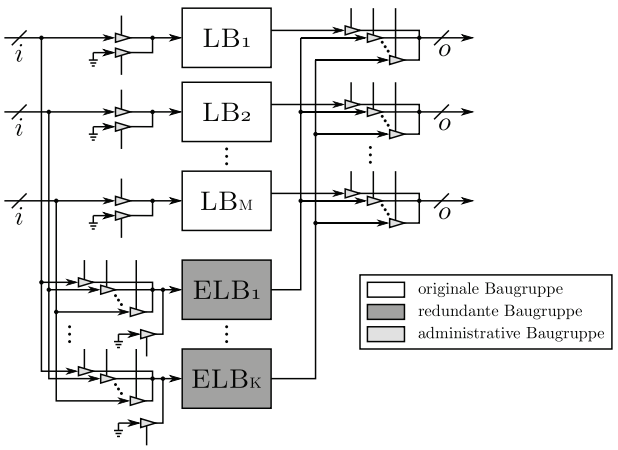


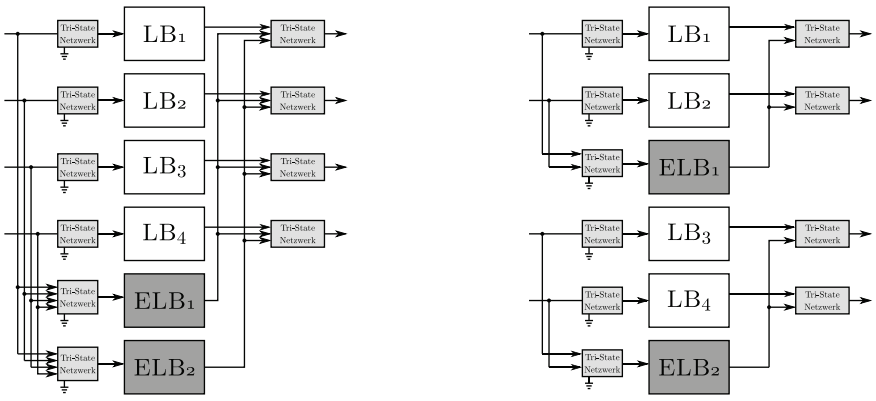
Abbildung 4.6: Implementierung eines Redundanten Logik-Blocks dieser Arbeit

4.1.4 Segmentierung der RLB

Der Aufbau der RLB Strukturen aus Abschnitt 4.1.1 wurde so gewählt, dass eine unterschiedliche Anzahl von Ersatz-Logik-Blöcken möglich ist. Diese Art der Implementierung bietet die Möglichkeit, eine optimale Redundanzstrategie für bestimmte Zwecke auszuwählen und einzusetzen. Die Anzahl der benötigten Tri-State Gatter für den RLB aus Abbildung 4.6 kann folgendermaßen errechnet werden:

$$\#\text{Tri-State Gatter} = 2I_{LB}M + I_{LB}K(M+1) + O_{LB}KM \tag{4.1}$$

Dabei steht I_{LB} und O_{LB} für die Anzahl der Ein- bzw. Ausgänge eines Logik-Blocks, M für die Anzahl der Logik-Blöcke und K für die Anzahl der Ersatz-Logik-Blöcke. Das Prinzip der Segmentierung der RLB ist in Abbildung 4.7 dargestellt. Die grundsätzliche Idee hier ist die dedizierte Verwendung einzelner Ersatz-Logik-Blöcke für eine Teilmenge der Logik-Blöcke. Beide RLB in Abbildung 4.7 haben die gleiche Anzahl ELB. Der RLB in Abbildung 4.7(a) kann beide ELB für beliebige LB ersetzen. Hingegen stehen dem System in Abbildung 4.7(b) nur jeweils ein ELB für zwei LB zur Verfügung.



(a) ein Segment (b) zwei Segmente

Abbildung 4.7: Redundanter Logik-Block mit unterschiedlichen Segmentierungen für $M = 4$ und $K = 2$

Durch eine Segmentierung werden zwar die Möglichkeiten des Austausches defekter Komponenten eingeschränkt, allerdings verringert sich der Aufwand für das Schaltnetzwerk.

$$\#\text{Tri-State Gatter} = P \cdot (2I_{LB}M_s + I_{LB}K_s(M_s + 1) + O_{LB}K_sM_s) \tag{4.2}$$

Mit Gleichung 4.2 lässt sich die Anzahl der benötigten Tri-State Gatter in den Multiplexern für unterschiedliche Segmentierungen berechnen. Hier beschreibt I_{LB} und O_{LB} wieder die Anzahl der Ein- und Ausgänge eines Logik-Blocks, P stellt die Anzahl der Segmente dar und M_s und K_s die Anzahl der Logik- und Ersatz-Logik-Blöcke des jeweiligen Segments. Die Auswirkungen des unterschiedlichen Wachstums des Schaltnetzwerkes sind in Abbildung 4.8 dargestellt. Hier ist zu sehen, dass bei gleicher Anzahl an Ersatz-Logik-Blöcken der Aufwand für die benötigten Schalter bei höherer Segmentierung geringer ist. Allerdings ist auch die Verwendung der Ersatz-Logik-Blöcke eingeschränkt, da ihre Verwendung nur im zugeteilten Segment möglich ist.

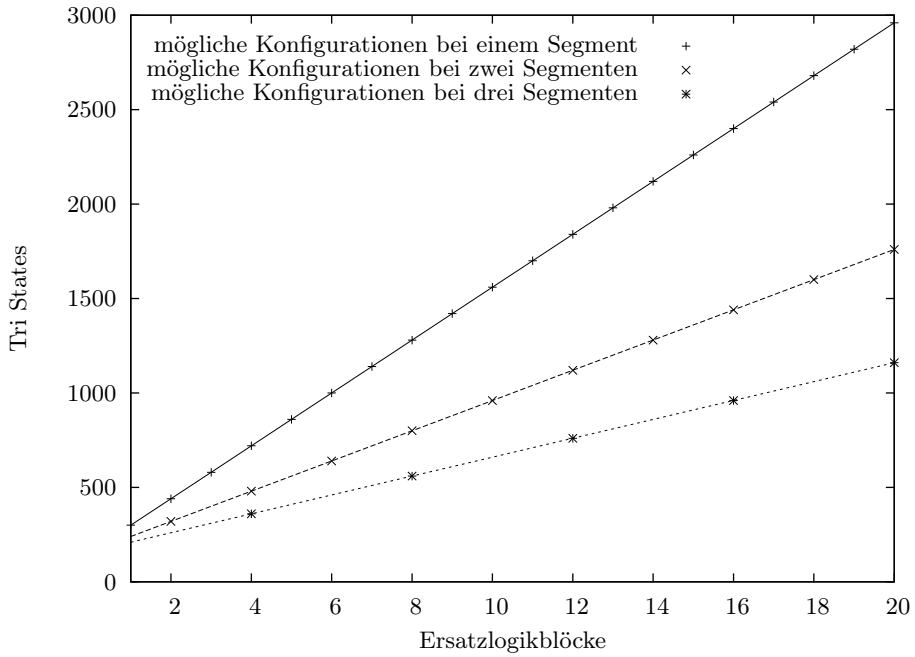


Abbildung 4.8: Anzahl der Tri-State Gatter für unterschiedliche Segmentierungen und Ersatz-Logik-Blöcke eines RLB mit $M = 4$,$I_{LB} = 20$,$O_{LB} = 10$ für $P = 1|2|4$

Die Auswirkung verschiedener Segmentierungen und die damit verbundene unterschiedliche Entwicklung des zusätzlichen Hardware-Mehraufwandes auf Zuverlässigkeit, Lebensdauer und Produktionsausbeute ist Teil der Entwurfsraumexploration dieser Arbeit und wird im nächsten Kapitel betrachtet.

4.1.5 Administration der RLB

Dieser Abschnitt beschreibt die Möglichkeiten der Administration der eben im Detail vorgestellten RLB. Administration meint hier die Verwaltung des RLB-Status, also die Steuerung der Rekonfiguration. Um eine vielseitige Verwendung der RLB sicherzustellen kommen nur Verfahren in Frage, welche die Administration vollständig *on-chip* realisieren. Nur dann kann dieser Ansatz auch im Feld angewendet werden. Essentiell sind hierfür dedizierte Speicherzellen, die den Zustand des RLB speichern.

	0	1	⋯	K
1	LB_1	ELB_1	⋯	ELB_K
2	LB_2	ELB_1	⋯	ELB_K
⋮	⋮	⋮	⋱	⋮
M	LB_M	ELB_1	⋯	ELB_K

Abbildung 4.9: Speicherfeld zur Administration eines RLB

Abbildung 4.9 zeigt das Prinzip der Verwendung der Speicherzellen zur Administration der RLB in dieser Arbeit. Hier wird jeder der $1..M$ Funktionen, die im RLB ausführbar sind, genau $0..K$ Speicherzellen zugewiesen. Es werden dann pro RLB genau $M \cdot (K+1)$ Speicherzellen benötigt. Durch eine bessere Zustandskodierung kann der Aufwand an Speicherzellen noch reduziert werden, was dann allerdings zu einer komplexeren Ansteuerfunktion der Tri-State Elemente führt. In der hier vorgestellten Implementierung kann jede Reihe des Speichers $1..M$ genau eine logische 1 beinhalten. Wobei eine logische 1 in den Speicherzellen $1..K$ das Ausführen der Funktion auf einem der K Ersatz-Logik-Blöcke bedeutet. Enthält die Speicherzelle 0 eine logische 1, dann wird die jeweilige Funktion auf dem ursprünglichen Logik-Block ausgeführt. In Abbildung 4.10 sind zwei mögliche Inhalte eines Speicherfeldes für einen RLB mit $M = 4$ und $K = 3$ dargestellt. Abbildung 4.10(a) zeigt die Ausgangskonfiguration des RLB. In dieser ist Spalte 0 vollständig mit logisch 1 initialisiert. Jede Funktion wird auf dem vorgesehenen

Logik-Block ausgeführt. Abbildung 4.10(b) zeigt den Zustand für den gleichen RLB der bereits Funktionen auf die ELB verschoben hat. Damit es keine doppelten Treiber auf ein oder mehrere Signale im RLB gibt, muss der Zustand des Speicherfeldes folgende Eigenschaften erfüllen:

- In jeder der M Zeilen des Speicherfeldes darf nur eine logische 1 stehen.
- In den Reihen $1..K$ eines Segments darf ebenfalls jeweils nur eine logische 1 stehen.

	0	1	2	3
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0
4	1	0	0	0

(a) initialer Zustand

	0	1	2	3
1	0	1	0	0
2	1	0	0	0
3	0	0	1	0
4	0	0	0	1

(b) modifizierter Zustand

Abbildung 4.10: Mögliche Zustände des RLB-Speichers mit $M = 4$ und $K = 3$

Damit eine Rekonfiguration des RLB stattfinden kann, benötigt die administrative Instanz sowohl lesenden als auch schreibenden Zugriff auf den hier vorgestellten RLB-Speicher. Der Schreibzugriff ermöglicht die eigentliche Rekonfiguration des RLB. Hingegen wird der Lesezugriff benötigt, um im Fehlerfall den nächsten freien ELB zu ermitteln, beziehungsweise eine erschöpfte Redundanz festzustellen. In Quellcode 4.1 findet sich dieser Algorithmus als Pseudocode.

Quellcode 4.1: Ermittlung des nächsten freien ELB

```
1  // Funktion zur Bestimmung des
2  // nächsten freien ELB
3  function nextELB($RLB_nr, $LB)
4  begin
5
6  // Initialisierung der neuen Konfiguration
7  $new_config(0 to K) = (others => '0')
8
9  // Bestimmen der Segmentgrenzen
10 $segment_begin = getSegmentBegin($RLB_nr, $LB)
11 $segment_end   =   getSegmentEnd($RLB_nr, $LB)
12
```

```
13 // Auslesen aller Konfigurationen des aktuellen RLB
14 for $i=$segment_begin to $segment_end loop
15   $new_config = $new_config OR readRLBConfig($RLB_nr, $i)
16 end loop
17
18 // Letzter ELB bereits aktiv?
19 if new_config(K) = 1 then
20   exception("Redundanz erschöpft!")
21   return null
22 else
23   for K-1 downto 1 loop
24     if new_config($i) == 0 and
25        new_config($i-1) == 1
26     then
27       // Initialisierung der neuen Konfiguration
28       new_config(0 to K) = (others => '0')
29       new_config($i) = 1
30       return new_config
31     end if
32   end loop
33 end if
34
35 end nextELB
```

Um im jeweiligen Fehlerfall für einen LB den nächsten freien ELB zu ermitteln, wird jeder Zustand des betroffenen RLB Segments mit einem logischen *oder* verknüpft. Enthält die höchstwertige Stelle K des verknüpften Wertes eine 1, so gilt die Redundanz dieses Segments als erschöpft. Falls die Redundanz noch nicht erschöpft ist, existiert noch mindestens ein ELB der noch nicht verwendet wird und als Ersatz in Frage kommt.

Weiterhin muss die administrative Instanz zusätzlich Test und Diagnose der einzelnen LB/ELB steuern, um gegebenenfalls eine Rekonfiguration des RLB vornehmen zu können. Im Quellcode 4.2 ist die Funktionsweise dieser Instanz beschrieben. Im folgenden wird diese Instanz stets als RLB-Controller bezeichnet.

Quellcode 4.2: Vollständiger Ablauf des RLB-Controller für Test, Diagnose und Rekonfiguration der RLB

```
1  // Test- und Rekonfigurationsfunktion
2  // der RLB-Controller-Komponente
3  function startReconfiguration()
4  begin
5
6    // für alle RLB
7    for $i=0 to #RLBs loop
8      $RLB_tested = false
9
10     // Teste bis RLB Funktionalität
```

```
11      // vollständig überprüft wurde
12      while $RLB_tested == false do
13
14        // für alle aktiven LB/ELB des RLB
15        for $j=0 to #LBs loop
16          $LB_tested = false
17
18          // Teste bis LB/ELB Funktionalität
19          // vollständig überprüft wurde
20          while $LB_tested == false do
21
22            // Test des aktuellen LB/ELB
23            if startTest($i,$j) == "PASS" then
24              $LB_tested = true
25            else
26              // Ermitteln des nächsten freien ELB
27              $new_config = nextELB($i,$j)
28              if $new_config == null then
29                exception("Redundanz erschöpft!")
30                return failed
31              end if
32              // Durchführen der Rekonfiguration
33              storeConfig($i,$j,$new_config)
34            end if
35          od
36        end loop
37        $RLB_tested
38      od
39    end loop
40    return passed
41  end startReconfiguration
```

Für jeden implementierten RLB wird durch den RLB-Controller ein Test gestartet und im Falle eines erkannten Fehlers eine Rekonfiguration durchgeführt. Zuerst wird der aktuelle Zustand des zu testenden RLBs ausgelesen. Anschließend folgt für jeden aktiven (Ersatz-)Logik-Block des RLB der Aufruf für den Test. Nachdem das Ergebnis des Tests verfügbar ist entscheidet sich, ob der RLB-Controller eine Rekonfiguration des aktuellen RLB durchführen muss. Bei einem erfolgreichen Test wird der nächste aktive (Ersatz-)Logik-Block getestet. Wird durch den Test ein Fehler im aktuellen LB oder ELB gefunden, erfolgt die Rekonfiguration des RLB (siehe Quelltext 4.1). Dies wird solange wiederholt, bis ein fehlerfreier Block gefunden wird oder die Redundanz des RLB erschöpft ist.

Verschiedene Systeme bieten unterschiedliche Möglichkeiten für die auslösenden Ereignisse der Ablaufsteuerung des RLB-Controllers. Die in dieser Arbeit umgesetzten Möglichkeiten werden in Abschnitt 4.3.1 im Detail erläutert.

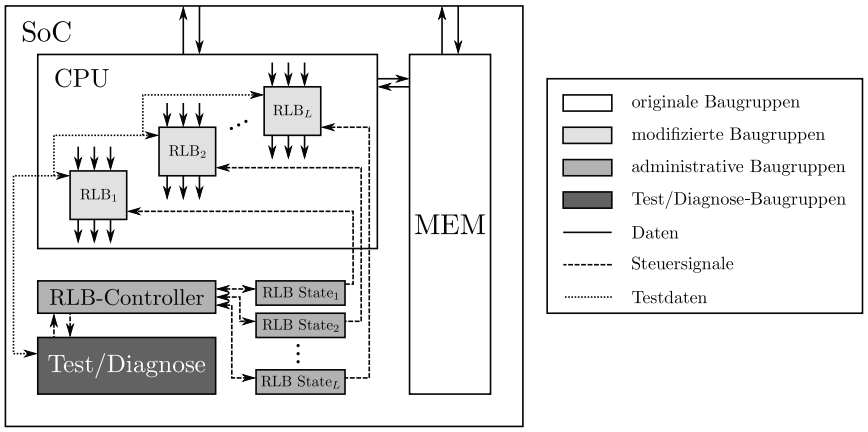


Abbildung 4.11: Übersicht einer modifizierten Architektur mit Möglichkeiten der Kompensation von permanenten Fehlern durch die RLB dieser Arbeit

In Abbildung 4.11 ist ein mögliches Gesamtsystem dargestellt. Dieses zeigt ein SoC bestehend aus CPU und Speicher. In der CPU wurden bestimmte Komponenten durch die vorher vorgestellten RLB ersetzt. Der Zustand jedes RLB wird in den zusätzlichen Komponenten RLB State$_1$ bis RLB State$_L$ gespeichert. Diese werden wiederum durch den zusätzlichen RLB-Controller gesteuert. Um den Anforderungen an den Ansatz nachzukommen, bezeichnet Test/Diagnose eine weitere Komponente auf dem SoC, die Test und Diagnose für die RLB im Feld realisieren muss. Die Möglichkeiten für einen Test und anschließende Diagnose auf dem Chip sind abhängig von der gewählten Architektur und werden im nächsten Abschnitt diskutiert. Grundsätzlich wurde der Ansatz so umgesetzt, dass das verwendete Testverfahren für den RLB-Controller transparent ist. Somit kann dieser Ansatz für unterschiedliche Architekturen, unabhängig von den jeweils zur Verfügung stehenden Möglichkeiten für Test und Diagnose, realisiert werden.

4.2 Fehlererkennung und Diagnose für permanente Fehler

Nachdem Aufbau, Funktion und Administration der RLB erläutert wurden, setzt sich dieser Abschnitt mit der Fehlererkennung und der Diagnose von permanenten Fehlern in den RLB auseinander. In Abschnitt 3.1.4.1 wurden bereits verschiedene Möglichkeiten vorgestellt, die eine Fehlererkennung für permanente Fehler leisten. Dabei wurde zwischen *online*- und *offline*-Verfahren zur Fehlererkennung unterschieden. Verfahren für die *online*-Fehlererkennung werden im folgenden nicht weiter berücksichtigt, da eine *online*-Fehlererkennung nicht Bestandteil dieser Arbeit ist.

	BIST (*built-in self-test*)
Vorteile	- sehr hohe Fehlerüberdeckung für statische Fehlermodelle - für beliebige Prozessor-Architekturen einsetzbar - sehr gute CAD Unterstützung (*push button*-Lösung) - benötigt lediglich strukturelle Informationen
Nachteile	- zusätzliche Hardware-Erweiterungen notwendig - beschränkter *at-speed* Test - unrealistische Belastung der Versorgungsspannung - führt zu *overtesting*
	SBST (*software-based self-test*)
Vorteile	- hohe Fehlerüberdeckung für statische Fehlermodelle in Komponenten des Datenpfades - keine Hardware-Erweiterungen notwendig - Schaltungstest im Arbeitstakt (*at-speed*-Test) - Verweidung von *overtesting* - realistische Belastung der Versorgungsspannung
Nachteile	- hohe Entwicklungskosten - zusätzlicher Speicherplatz für die Test-Software notwendig - aufwendige Ermittlung einer strukturellen Fehlerüberdeckung - benötigt zusätzliche Informationen der Architektur

Tabelle 4.1: Gegenüberstellung von BIST- und SBST-Verfahren zur Erkennung von permanenten Fehlern im Feld

Verfahren für die *offline* Fehlererkennung sind zum ersten der eingebaute Selbsttest (BIST) und zum anderen der softwarebasierte Selbsttest (SBST). In Tabel-

le 4.1 findet sich eine Gegenüberstellung beider Verfahren mit ihren jeweiligen Vor- und Nachteilen.

Die Anwendbarkeit eines Test- und Diagnoseverfahrens im Feld ist eine notwendige Anforderung für den Ansatz dieser Arbeit. Weitere Anforderungen an Test und Diagnose lassen sich wie folgt zusammenfassen:

- Das verwendete Testverfahren muss eine hohe Fehlerüberdeckung für die Logik- und Ersatz-Logik-Blöcke der RLB erreichen.
- Die Diagnosefähigkeit muss mindestens eine *pass/fail* Analyse für die einzelnen LB und ELB liefern.
- Der zusätzliche Hardware-Aufwand für das Test- und Diagnoseverfahren sollte möglichst gering sein, da dieser die Effizienz des Ansatzes verringert.

Die Realisierung eines bestimmten Test- und Diagnoseverfahren ist für jeden konkreten Einsatz zu prüfen. Eine allgemeingültige Entscheidung für die Verwendung einer speziellen Methode für Test und Diagnose lässt sich aus dieser Arbeit nicht ableiten. Daher wurde bei der Umsetzung des Aufbaus und der Administration der RLB in Abschnitt 4.1 darauf geachtet, dass Test- und Diagnoseverfahren problemlos auszutauschen sind und lediglich die Funktionalität des RLB-Controller modifiziert werden muss. In den nächsten Abschnitten 4.2.1 und 4.2.2 wird auf die Details der Umsetzung für BIST- und SBST-Verfahren eingegangen.

4.2.1 Test und Diagnose mittels BIST

Die Umsetzung von Test und Diagnose mit Hilfe eines BIST ist eine Möglichkeit zur Realisierung des hier vorgestellten Ansatzes der Rekonfiguration. In Abbildung 4.12 ist dargestellt, wie ein BIST im System integriert werden muss. Für die überwachende Instanz bleibt das Testverfahren transparent.

Für den Ablauf von Test und Rekonfiguration muss der BIST durch den RLB-Controller gestartet werden (Quellcode 4.2 (Zeile 21)). Nach Ablauf von Test und Diagnose muss der BIST dem RLB-Controller lediglich die Information über einen erfolgreichen beziehungsweise nicht erfolgreichen Test zur Verfügung stellen.

Eine Realisierung von Test und Diagnose sollte vor allem dann durch ein BIST bevorzugt werden, wenn es sich bei dem verwendeten Prozessor um eine dynamisch geplante Architektur handelt. Genau in diesem Fall ist die Entwicklung

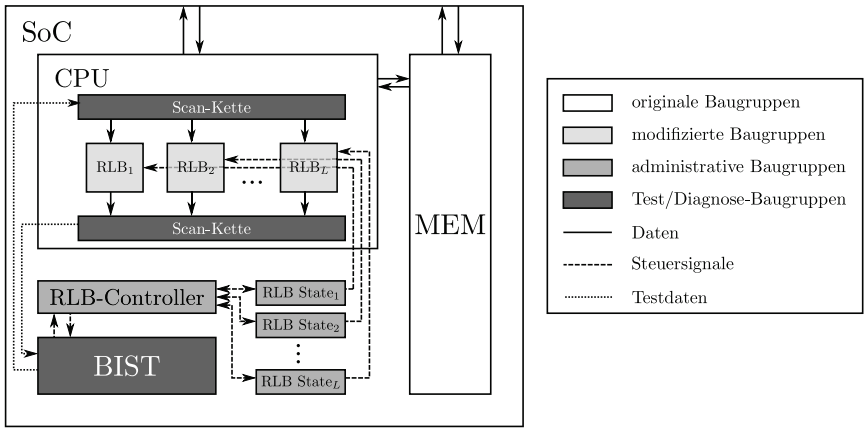


Abbildung 4.12: Architektur mit BIST als Testverfahren für die RLB

einer effizienten Testsoftware aufwendiger, da die dynamische Ablaufplanung besondere Einschränkungen für die Erstellung eines SBST bedeutet. Gleichzeitig kann der BIST problemlos Test und Diagnose parallel für alle funktionalen Einheiten realisieren, denn die notwendigen Testmuster können durch Schiebepfade direkt an die zu testenden Einheiten gelangen und die Testantworten liefern aufgrund ihrer Zuordnung eine bessere Diagnose. Somit kann die Ablaufsteuerung des RLB-Controller weiter reduziert werden. Auch sequentielle Teilschaltungen in den RLB stellen kein Problem für ein BIST basiertes Verfahren dar, da durch die Verwendung von Scan-Ketten keine sequentiellen Testmuster notwendig sind.

4.2.2 Test und Diagnose mittels SBST

Eine mögliche Alternative zu den gängigen BIST-Verfahren ist die Umsetzung mittels eines softwarebasierten Selbsttests. Abbildung 4.13 zeigt die Integration dieses Test- und Diagnoseverfahrens für die vorgestellten RLB. Auch diese Implementierung bleibt für die überwachende Instanz transparent.

Der Ablauf von Test und Rekonfiguration ist identisch zu der Implementierung mit einem BIST. Der SBST stellt immer dann eine mögliche Alternative dar, wenn es sich bei der Zielarchitektur um einen Prozessor mit einer statischen Ablaufplanung handelt. In diesem Fall ist die Entwicklung eines geeigneten SBST einfacher zu realisieren, weil er weniger Einschränkungen unterliegt. Für einen softwarebasierten Selbsttest ist allerdings zu beachten, dass ein paralleler Test aller funktionaler Einheiten zwar realisiert werden kann, die notwendige dia-

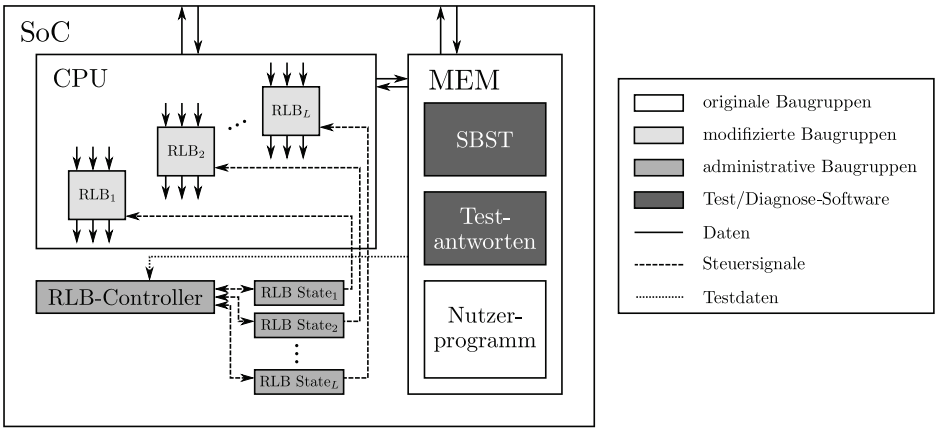


Abbildung 4.13: Architektur mit SBST als Testverfahren für die RLB

gnostische Auflösung aufgrund der Ausführung einer Software kann allerdings verloren gehen. Grundsätzlich ist auch eine Erstellung eines SBST für sequentielle Schaltungen möglich. Abhängig von der verwendeten Architektur können die funktionalen Einschränkungen, welchen der SBST unterliegt, ein deutliches Absenken der Fehlerüberdeckung des Tests bewirken.

Für das Anwendungsbeispiel dieser Arbeit in Kapitel 6 wird als Test- und Diagnoseverfahren der SBST verwendet. Folgende Auflistung enthält die maßgeblichen Gründe für diese Entscheidung:

- Eine SBST-Lösung benötigt keine weitere Hardware bis auf den zusätzlichen Speicherplatz für die Software und deren Testmuster.
- SBST-Lösungen liefern vor allem für funktionale Einheiten dem BIST-Verfahren vergleichbar hohe Fehlerüberdeckungen [PGSR10].
- Durch SBST-Verfahren wird die Schaltung im funktionalen Modus betrieben, welcher die Schaltung unter realen Bedingungen (*at-speed* und realistische Belastungen der Versorgungsspannung) testet.
- Durch eine nachträgliche Modifikation der Software lässt sich die Test- und Diagnosefähigkeit an bestimmte Fehlersituationen anpassen.
- Das Anwendungsbeispiel dieser Arbeit verfügt über eine statische Ablaufplanung und eignet sich daher besonders für die Umsetzung eines SBST.

4.3 Funktionsweise des Gesamtsystems

Dieser Abschnitt erläutert den zeitlichen Ablauf des Gesamtsystems zur Kompensation von permanenten Fehlern für den weiteren Betrieb. Der verfolgte Ansatz ist grundsätzlich nicht in der Lage, transiente oder intermittierende Fehler zu erkennen oder zu kompensieren. Dafür sind zusätzliche Verfahren der Fehlertoleranz speziell für diese Fehler notwendig. Auch auftretende permanente Fehler werden nicht *online* erkannt und kompensiert.

Durch den Ansatz dieser Arbeit wird verhindert, dass nach der Produktion existierende und im Einsatz auftretende permanente Fehler zu einem dauerhaften Fehlverhalten des Systems führen. In Abbildung 4.14 ist der Ablauf des Gesamtsystems dargestellt. Das Ablaufdiagramm besteht aus zwei Phasen. In der ersten ist der Ablauf nach einem Systemstart abgebildet. Die zweite Phase zeigt den Ablauf des Systems im Anwendungsfall.

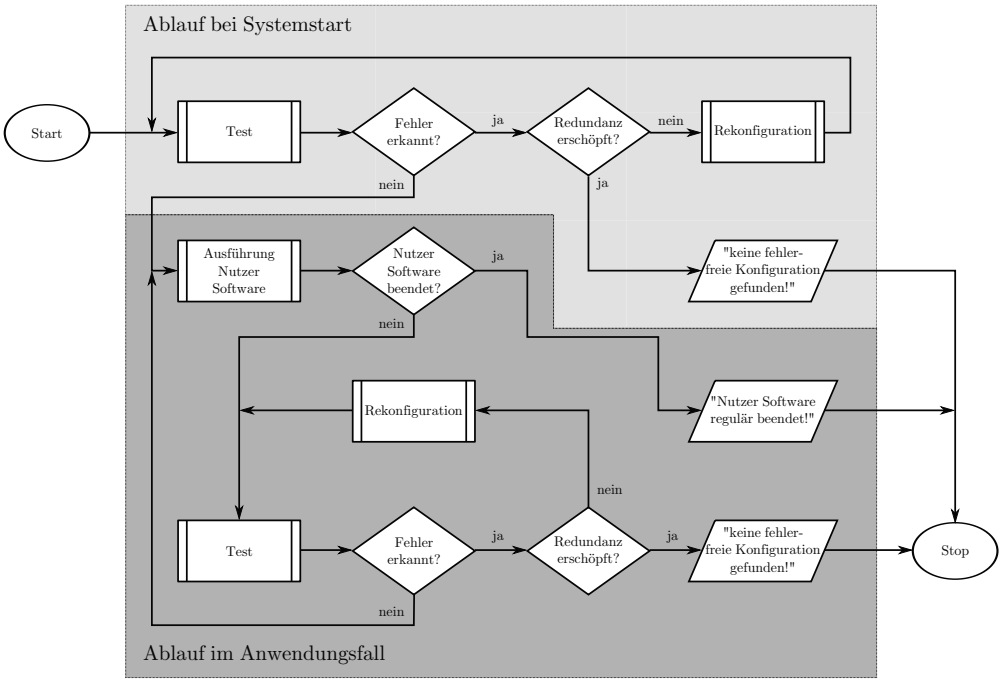


Abbildung 4.14: Ablaufdiagramm des Gesamtsystems zur Kompensation permanenter Fehler nach dem Systemstart und im Anwendungsfall

Nach der Inbetriebnahme übernimmt der RLB-Controller die Kontrolle über das Gesamtsystem und veranlasst, wie bereits in Abschnitt 4.1.5 beschrieben,

einen Test der einzelnen redundant ausgelegten Komponenten. Findet der Test eine fehlerhafte Komponente, so wird eine Rekonfiguration des Systems durchgeführt und der Test wiederholt. Führt eine wiederholte Rekonfiguration zu einer erschöpften Redundanz, so wird das System mit entsprechender Fehlermeldung angehalten.

Nachdem ein fehlerfreier Zustand durch den Test bestätigt wurde folgt der funktionale Systembetrieb und damit die Ausführung der Nutzersoftware. Um auf mögliche Alterungseffekte zu reagieren, wird die Abarbeitung der Nutzersoftware zyklisch unterbrochen und das System wiederholt einem Test unterzogen, welcher bei erkannten Fehlern wieder eine Rekonfiguration veranlasst und anschließend den funktionalen Betrieb fortsetzt. Auch während des zyklischen Tests hat der RLB-Controller die vollständige Kontrolle über das Gesamtsystem.

4.3.1 Zeitlicher Ablauf

Aus Abbildung 4.14 geht hervor, dass der Test der Hardware zu zwei unterschiedlichen Zeitpunkten realisiert werden muss. Dazu zählt zum ersten der Test nach dem Systemstart und zum zweiten die zyklischen Tests im laufenden Betrieb. Für die verschiedenen Zeitpunkte der Ausführung von Test, Diagnose und Rekonfiguration ergeben sich unterschiedliche Konsequenzen für die Verfügbarkeit des Systems.

Rekonfiguration beim Systemstart

Der Test mit möglicher Rekonfiguration der RLB nach dem Systemstart führt lediglich zu einer Verzögerung der Einsatzbereitschaft des Systems. In diesem Fall beginnt die Abarbeitung des Nutzerprogrammes erst nachdem durch Test und Rekonfiguration aller L RLB ein fehlerfreier Zustand gefunden wurde. Diese zusätzliche Zeit t_{start} ist bei einem fehlerfreien System konstant und kann wie folgt berechnet werden:

$$t_{start} = \sum_{i=1}^{L} M_i(t_{\mathrm{test}_i} + t_{\mathrm{control}_i}) \tag{4.3}$$

Im fehlerfreien Fall sind dann für jeden der i RLB genau M_i LB zu testen. Diese Testzeit setzt sich zusammen aus der Ablaufzeit des Tests t_{test_i} und der Zeit zur Vorbereitung und Auswertung des Tests durch den RLB-Controller t_{control_i} für den zu testenden LB/ELB.

Im Fehlerfall ist die Verzögerung der Einsatzbereitschaft des Systems abhängig von Anzahl und Ort der Fehler. Im Fehlerfall lässt sich eine obere Schranke für die Verzögerung bestimmten:

$$t_{start\_max} = \sum_{i=1}^{L} (M_i + K_i)(t_{\text{test}_i} + t_{\text{control}_i}) + K_i \cdot t_{\text{reconf}_i} \tag{4.4}$$

Diese obere Schranke ergibt sich genau dann, wenn für jeden der i RLB der fehlerfreie Zustand der letzt mögliche Zustand vor einer erschöpften Redundanz ist. In diesem Fall ergibt sich für jeden der $M+K$ LB/ELB eines RLB die zusätzliche Zeit t_{test_i} für die Ausführung des Tests und die Zeit t_{control_i} für Vorbereitung und Auswertung des Tests. Zusätzlich fällt die Zeit t_{reconf_i} für jede der K_i Rekonfigurationen jedes RLB an.

Für den konkreten Einsatz dieses Verfahrens ist zu klären, ob diese obere Schranke für die Verzögerung der Einsatzbreitschaft des Systems tragbar ist. Weitere Implikationen des Tests zum Systemstart erbeben sich nicht.

Rekonfiguration im Anwendungsfall

Die in Abbildung 4.14 dargestellten zyklischen Tests im laufenden Betrieb werden in dieser Arbeit durch zwei unterschiedliche Ereignisse ausgelöst. Im ersten Fall übergibt die Anwendersoftware die Kontrolle des Gesamtsystems an den RLB-Controller. Diese geplante Ausführung von Test und Diagnose kann beispielsweise in Leerlaufphasen durchgeführt werden.

Im zweiten Fall wird der Test durch den RLB-Controller erzwungen. Dies stellt eine ungeplante Prüfung des Systems dar und wird genau dann benötigt, wenn auftretende Fehler dazu führen, dass die Anwendersoftware die Kontrolle nicht mehr an den RLB-Controller übergeben kann.

Beide Varianten führen dazu, dass das System für die Zeit der Prüfung nicht verfügbar ist. Jedoch beschränkt sich diese Zeit nicht nur auf die in Gleichungen 4.3 und 4.4 beschriebenen Zeiten für den Test und eine mögliche Rekonfiguration des Systems. Zusätzlich dazu ist bei den zyklischen Tests eine Sicherung und anschließende Wiederherstellung des Prozessorstatus notwendig, weil es während des Tests zu einer vollständigen Abschaltung der CPU kommt.

Somit ergibt sich eine Gesamtzeit $t_{im\ Feld}$ (Gleichung 4.5) in der das System nicht verfügbar ist, welche sich aus der notwendigen Zeit t_{save} zur Sicherung und der Zeit $t_{restore}$ zur Wiederherstellung des Prozessorstatus sowie der benötigten Zeit t_{start} beziehungsweise t_{start_max} für den Test und die Rekonfiguation der RLB

zusammensetzt.

$$t_{im\ Feld} = t_{save} + t_{(start|start\_max)} + t_{restore} \tag{4.5}$$

Das Sichern und Wiederherstellen des Prozessorstatus dient dazu, nach einem zyklischen Test, welcher keinen Fehler findet und demzufolge keine Rekonfiguration der RLB veranlasst, die Nutzerroutine fortzusetzen.

Im Falle eines gefundenen Fehlers durch den zyklischen Test, welcher zur Rekonfiguration von mindestens einem RLB führt, ist eine Fortsetzung der Nutzersoftware nicht möglich. Hier ist davon auszugehen, dass der gefundene Fehler bereits den Systemzustand kompromittiert hat und damit die Wiederherstellung des Systemzustands und die anschließende Fortsetzung der Nutzersoftware zu fehlerhaften Ergebnissen führt. Aus diesem Grund folgt nach einer Rekonfiguration im laufenden Betrieb ein Neustart des Systems. Durch das Erstellen von Checkpunkten kann auch in diesen Fällen eine Fortsetzung der Nutzersoftware realisiert werden. Diese müsste den letzten validen Checkpunkt als Wiederherstellungspunkt wählen.

Der zeitliche Ablauf des vorgestellten Ansatzes lässt sich bezüglich der auslösenden Ereignisse für den Test durch bekannte Techniken erweitern. Beispielsweise können typische Maßnahmen der Fehlertoleranz für transiente oder intermittierende Fehler, wie TMR oder ECC, Fehlersignale erzeugen die dann als auslösendes Ereignis für den Test dienen. Ebenfalls können Signale von *on-chip* Sensoren zur Feststellung von Parameterverschiebungen [AVU+07] im laufenden Betrieb einen zyklischen Test auslösen.

Weitere Möglichkeiten bestehen durch die Nutzung von Compilern, die fehlertolerante Software erzeugen [Sch10]. Hier kann die Software typische Fehlersymptome oder Verstöße erkennen, indem zusätzliche redundante Instruktionen ausgeführt werden [DQ11]. Die Kombination von Compilern, die fehlertolerante Programme erstellen, mit Hardware-Erweiterungen sind ebenfalls dazu geeignet vor allem Kontrollfluss-Fehler zu erkennen [DQ11]. Alle diese compilerbasierten Techniken können ebenfalls dazu dienen, den Test des Systems zu veranlassen.

Die hier vorgestellten möglichen Erweiterungen bedeuten keine weiteren als die bereits vorgestellten Einschränkungen bezüglich des zeitlichen Systemverhaltens mit sich bringen. Eine schnellere Erkennung von Fehlern im laufenden Betrieb bedeutete auch eine schnellere Anpassung an spezifische Fehlersituationen.

4.3.2 Skalierbarkeit des Ansatzes

Nachdem in den vorherigen Abschnitten der Aufbau und die Funktionsweise der RLB, sowie deren Verwaltung einschließlich Test und Diagnose erläutert wurden, folgt in diesem Abschnitt eine Betrachtung der Möglichkeiten zur Skalierung des Ansatzes.

Durch den vorgestellten Aufbau der RLB ist es möglich, jede Funktion eines Prozessors, repräsentiert durch einen Logik-Block, um eine beliebige Anzahl von Ersatz-Logik-Blöcken zu erweitern. Ebenfalls erlaubt der gewählte Aufbau der RLB die Zusammenfassung mehrerer identischer Funktionen, welche sich somit eine beliebige Anzahl von Ersatz-Logik-Blöcken teilen. Der entstehende Aufwand für das benötigte Schaltnetzwerk zum Umleiten der Funktion steigt linear zur Anzahl der verwendeten Ersatz-Logik-Blöcke (siehe Abbildung 4.8). Bei einer Segmentierung der RLB steigt der Aufwand für das Schaltnetzwerk ebenfalls linear zur Anzahl der verwendeten Ersatz-Logik-Blöcke jedoch geringer als bei einem RLB ohne Segmentierung.

Der notwendige zeitliche Aufwand für den Test und eine anschließende Rekonfiguration steigt ebenfalls linear zur Anzahl der verwendeten Ersatz-Logik-Blöcke im jeweiligen RLB. Eine Umsetzung der vorgestellten RLB ist ebenfalls als hierarchische Struktur möglich. Dazu würden die einzelnen Logik-Blöcke eines RLB aus weiteren RLB zusammensetzt (siehe Abbildung 4.15).

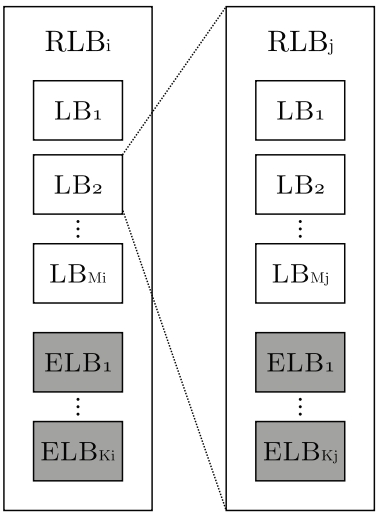


Abbildung 4.15: Hierarchische Implementierung der RLB

Im Fall eines hierarchischen Aufbaus der RLB ändert sich jedoch der zeitliche Aufwand für Test, Diagnose und Rekonfiguration, da eine erschöpfte Redundanz der inneren RLB (RLB_j in Abbildung 4.15) nicht zwingend zu einer erschöpften Redundanz der äußeren RLB (RLB_i in Abbildung 4.15) führt.

4.4 Arbeitsablauf

In diesem letzten Abschnitt des Kapitels wird gezeigt, wie der vorgestellte Ansatz in einen bestehende Entwurfsprozess zu integrieren ist.

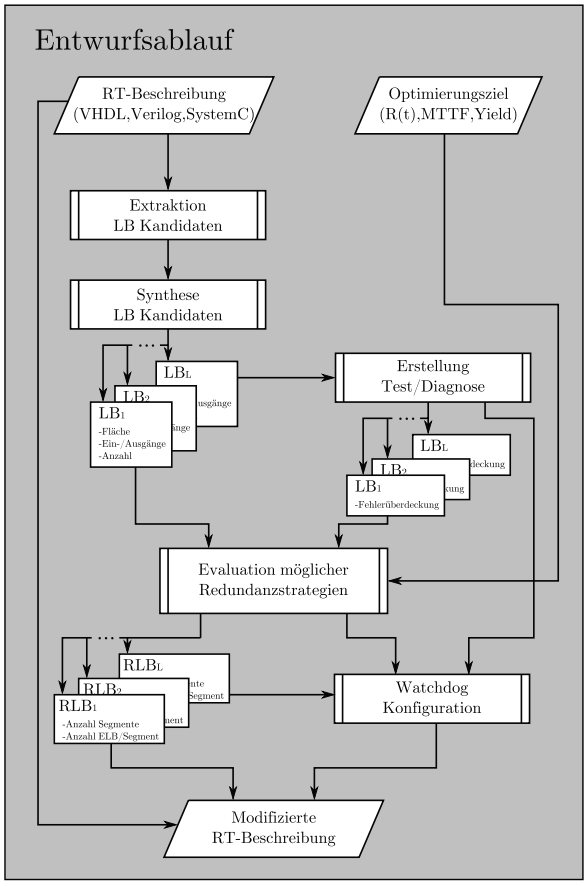


Abbildung 4.16: Arbeitsablauf zur Umsetzung und Integration des vorgestellten Ansatzes in einen bestehenden Entwurfsprozess

Zur Realisierung des hier vorgestellten Ansatzes ist der in Abbildung 4.16 gezeigte Arbeitsablauf zu durchlaufen. Eingabedaten sind zum einen die RT-Beschreibung einer Architektur und zum anderen ein Ziel, welches durch den Einsatz von redundanten Baugruppen für diese Architektur erreicht werden soll. Als Zielvorgaben kommen eine Steigerung der Ausbeute der Produktion, eine Steigerung der mittleren Lebensdauer und auch eine Erhöhung der Zuverlässigkeit in Frage.

In einem ersten Schritt sind mögliche LB Kandidaten aus der RT-Beschreibung der Architektur zu extrahieren. Anschließend folgt eine Synthese dieser Kandidaten um notwendige Eigenschaften wie Fläche, Anzahl benötigter Ein- und Ausgänge sowie die Anzahl identischer Komponenten zu ermitteln. Anschließend ist für jede Menge von identischen LB ein Test- und Diagnoseverfahren auszuwählen, um die erreichbare Fehlerüberdeckung der einzelnen Logik-Blöcke zu ermitteln. Danach folgt eine Evaluierung möglicher Redundanzstrategien anhand der ermittelten Eigenschaften der einzelnen LB und dem geforderten Ziel der Optimierung. Diese Evaluation wird im nächsten Kapitel detailliert beschrieben. Nach der Evaluation stehen die optimierten Konfigurationen für die jeweiligen RLB und den RLB-Controller fest und sind in die RT Beschreibung zu integrieren. Anschließend können weitere Schritte im Entwurfsprozess durchlaufen werden.

KAPITEL 5

Optimierte Redundanz-Auswahl

Nachdem im vorherigen Kapitel der Aufbau und die Funktionsweise der Rekonfiguration für Prozessorkomponenten vorgestellt wurde, folgt in diesem Kapitel die Systemmodellierung zur effizienten Auswahl der RLB-Konfigurationen. Diese Entwurfsraumexploration liefert für ein gegebenes Ausgangssystem eine optimierte Konfiguration. Mögliche Ziele der Optimierung sind das Maximieren der Zuverlässigkeit, der mittleren Lebensdauer und der Produktionsausbeute unter Berücksichtigung einer Obergrenze zusätzlich benötigter Hardware.

5.1 Modellierung des Systems

5.1.1 Modellierung der Zuverlässigkeit

Zur Ermittlung der Zuverlässigkeit wird in dieser Arbeit das bereits in Abschnitt 2.2.1 vorgestellte *reliability block diagram* verwendet. Das gesamte Design eines ICs wird dafür in eine bestimmte Anzahl von Einheitsflächen eingeteilt. Die Größe eines Flächenequivalents ist identisch mit der Größe der kleinsten Zelle der verwendeten Technologiebibliothek. In Abbildung 5.1 ist die Einteilung in Flächen des ursprünglichen Designs sowie des modifizierten Designs dargestellt.

Die Zuverlässigkeit $R_a(t)$ der Einheitsfläche a wird durch eine Verteilungsfunktion dargestellt, welche hier exemplarisch die Exponentialverteilung mit der konstanten Fehlerrate λ ist.

$$R_a(t) = e^{-\lambda \cdot t} \tag{5.1}$$

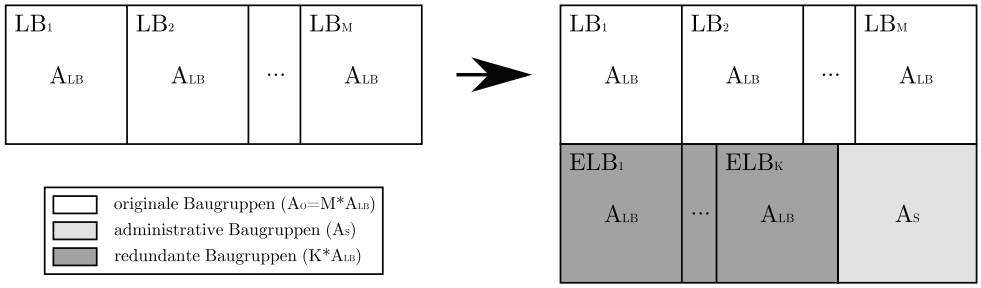


Abbildung 5.1: Flächenaufteilung des originalen (*links*) und des modifizierten Systems (*rechts*)

Ein komplexes System mit einer Vielzahl an Einheitsflächen kann dann durch eine Serienkomposition folgendermaßen beschrieben werden:

$$R_O(t) = R_a(t)^{A_O} \tag{5.2}$$

Hier wird $R_O(t)$ als die Zuverlässigkeit eines Seriensystems bestehend aus A_O Einheitsflächen dargestellt. Damit stellt Gleichung 5.2 die Zuverlässigkeit des Ausgangssystems O dar, welches keine Möglichkeiten der Fehlerkompensation beinhaltet. Vereinfachend wird hier angenommen, dass sich das Ausgangssystem O aus genau M identischen Logik-Blöcken (LB) mit jeweils der Fläche A_{LB} zusammensetzt (Gleichung 5.3).

$$A_{LB} = \frac{A_O}{M} \tag{5.3}$$

Wird jetzt das Ausgangssystem mit M LB zu den im letzten Kapitel vorgestellten RLBs um K Ersatz-Logik-Blöcke (ELB) erweitert, dann lässt sich die Zuverlässigkeit $R_{RLB}(t)$ für dieses modifizierte System folgendermaßen beschreiben:

$$R_{RLB}(t) = R_S(t) \cdot R_{M \text{ aus } N}(t) \tag{5.4}$$

Die Zuverlässigkeit von $R_{M \text{ aus } N}(t)$ beinhaltet die Anzahl M der ursprünglichen LB und die Anzahl $N = M + K$ aller LB/ELB einer bestimmten RLB-Konfiguration. Alle zusätzlichen Hardwarekomponenten, die zur Administration und zum Test benötigt werden, sind durch die Zuverlässigkeit $R_S(t)$ beschrieben. Diese stellt wieder ein Seriensystem für die Fläche A_S dar (Gleichung 5.5).

$$R_S(t) = R_a(t)^{A_S} \tag{5.5}$$

Das modifizierte System, in diesem Fall ein RLB, funktioniert demnach so lange, wie mindestens M LB oder ELB und die zusätzlichen administrativen Komponenten der Schaltung A_S keinen Fehler aufweisen.

$$R_{M \text{ aus } N}(t) = \sum_{i=M}^{N} \binom{N}{i} R_{LB}(t)^i \cdot [1 - R_{LB}(t)]^{N-i} \tag{5.6}$$

In Gleichung 5.6 ist die Zuverlässigkeit $R_{M \text{ aus } N}(t)$ dargestellt. Hier steht $R_{LB}(t)$ für die Zuverlässigkeit eines (Ersatz-)Logik-Blocks. Diese Zuverlässigkeit ist ebenfalls als Seriensystem zu verstehen und ergibt sich demnach wie folgt:

$$R_{LB}(t) = R_a(t)^{A_{LB}} \tag{5.7}$$

Demzufolge ergibt sich die Zuverlässigkeit $R_{RLB}(t)$ eines RLB durch Gleichung 5.8.

$$R_{RLB}(t) = R_a(t)^{A_S} \cdot \sum_{i=M}^{N} \binom{N}{i} R_a(t)^{i \cdot A_{LB}} \cdot [1 - R_a(t)^{A_{LB}}]^{N-i} \tag{5.8}$$

In Abbildung 5.2 ist die Zuverlässigkeit für das ursprüngliche System $R_O(t)$ sowie für das modifizierte System $R_{RLB}(t)$ mit einem Logik-Block $M = 1$ und einem Ersatz-Logik-Block $K = 1$ für verschiedene Verhältnisse von $r = \frac{A_S}{A_O}$ dargestellt. Hier wird ersichtlich, dass eine deutliche Steigerung der Zuverlässigkeit $R_{RLB}(t)$ gegenüber $R_O(t)$ für kleine Verhältnisse der Flächen $\frac{A_S}{A_O}$ möglich ist. Bei einem Anstieg des Flächenverhältnisses wird die Steigerung der Zuverlässigkeit $R_{RLB}(t)$ gegenüber $R_O(t)$ immer geringer und kann sogar zu einer Verschlechterung führen.

Die obere Schranke für die Steigerung der Zuverlässigkeit $R_{RLB}(t)$ gegenüber $R_O(t)$ ist die Zuverlässigkeit $R_{M \text{ aus } N}(t)$ für den Fall, dass die Fläche der zusätzlichen Hardware $A_S = 0$ ist. In diesen Fall wäre die Steigerung der Zuverlässigkeit gegenüber $R_O(t)$ maximal. Denn aus der Eigenschaft des Serien-Systems in Gleichung 5.8 folgt:

$$R_{RLB}(t, A_S = 0) \geq R_{RLB}(t, A_S > 0) \tag{5.9}$$

Die untere Schranke für die Steigerung der Zuverlässigkeit $R_{RLB}(t)$ ist dann erreicht, wenn die zusätzliche nicht redundante Fläche A_S die Zuverlässigkeit $R_{RLB}(t)$ dominiert und es folglich keinen Wert für $t > 0$ gibt, für den gilt $R_{RLB}(t) > R_O(t)$. Abbildung 5.2 lässt vermuten, dass die untere Schranke durch das Verhältnis von $r = \frac{A_S}{A_O}$ bestimmt wird.

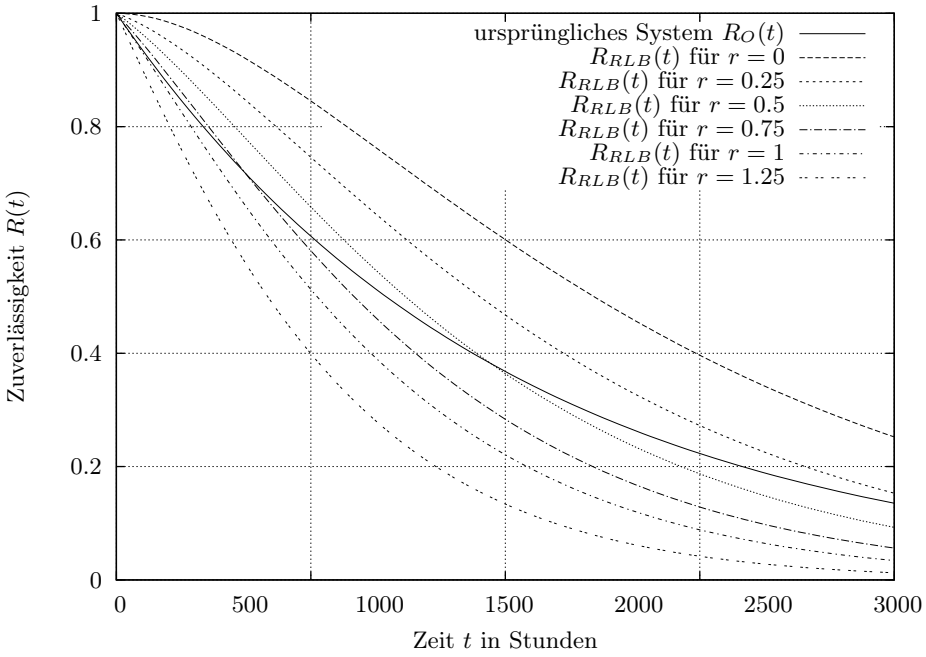


Abbildung 5.2: Entwicklung der Zuverlässigkeit $R_O(t)$ und $R_{RLB}(t)$ für unterschiedliche Flächenverhältnisse $r = \frac{A_S}{A_O}$ für ein System mit $M = 1, N = 2, A_O = 10^6$ und einer Fehlerrate $\lambda_a = 10^{-9}$ für ein Flächenequivalent.

Für $r \geq 1$ und $t > 0$ gilt:

$$R_a(t)^{A_O} > R_a(t)^{A_S} \tag{5.10}$$

und somit gilt ferner:

$$R_a(t)^{A_O} > R_a(t)^{A_S} \cdot R_{M \text{ aus } N}(t) \tag{5.11}$$

Demzufolge ist eine Steigerung der Zuverlässigkeit von $R_{RLB}(t)$ gegenüber $R_O(t)$ für Flächenverhältnisse $r \geq 1$ ausgeschlossen. Dies gilt unabhängig von der Zuverlässigkeit $R_{M \text{ aus } N}(t)$, da auch diese Zuverlässigkeit keine Werte größer 1 annehmen kann.

Um zu zeigen, dass es für Flächenverhältnisse $r < 1$ eine Steigerung der Zuverlässigkeit geben kann, soll folgend gezeigt werden, dass der Anstieg der Funktion $R_{RLB}(0)$ größer ist als der Anstieg von $R_O(0)$. Genau dann würde die Zuverlässigkeit $R_{RLB}(t)$ gegenüber der Zuverlässigkeit $R_O(t)$ für Werte von $t \to 0$ einen höheren Wert besitzen. Dazu sind die Funktionen der Zuverlässigkeit $R_O(t)$ und

$R_{RLB}(t)$ zu differenzieren. Dafür wird für die Zuverlässigkeit $R_a(t)$ die Exponentialverteilung ersetzt.

$$R_O(t) = e^{-\lambda t A_O} \tag{5.12}$$

Die Ableitung $\frac{dR_O(t)}{dt}$ für das ursprüngliche System ist folgend dargestellt:

$$R'_O(t) = -A_O \cdot \lambda \cdot e^{-\lambda t A_O} \tag{5.13}$$

Damit ist der Anstieg der Funktion zum Zeitpunkt $t = 0$:

$$R'_O(0) = -A_O \cdot \lambda \tag{5.14}$$

Nach Ersetzen der Zuverlässigkeit $R_a(t)$ durch die Exponentialverteilung und Auflösen der Summe für die Summanden $i = N$ und $i = N - 1$ erhält man:

$$\begin{aligned} R_{RLB}(t) = e^{-\lambda t A_S} \cdot \Bigg(N! \sum_{i=M}^{N-2} \left(\frac{e^{-\lambda t A_{LB} i} [1 - e^{-\lambda t A_{LB}}]^{N-i}}{i!(N-i)!} \right) \\ + \frac{N! e^{-\lambda t A_{LB}(N-1)} [1 - e^{-\lambda t A_{LB}}]}{(N-1)!} + e^{-\lambda t A_{LB} N} \Bigg) \end{aligned} \tag{5.15}$$

Die Ableitung $\frac{dR_{RLB}(t)}{dt}$ für das modifizierte System ergibt dann:

$$\begin{aligned} R'_{RLB}(t) = e^{-\lambda t A_S} \cdot \Bigg(N! \sum_{i=M}^{N-2} \Bigg(\frac{\lambda A_{LB}(N-i) e^{-\lambda t A_{LB} i - \lambda t A_{LB}} [1 - e^{-\lambda t A_{LB}}]^{N-i-1}}{i!(N-i)!} \\ - \frac{\lambda A_{LB} i e^{-\lambda t A_{LB} i} [1 - e^{-\lambda t A_{LB}}]^{N-i}}{i!(N-i)!} \Bigg) + \frac{\lambda A_{LB} N! e^{-\lambda t A_{LB}(N-1) - \lambda t A_{LB}}}{(N-1)!} \\ - \lambda A_{LB} N e^{-\lambda t A_{LB} N} - \frac{\lambda A_{LB} N! (N-1) e^{\lambda t A_{LB}(N-1)} [1 - e^{-\lambda t A_{LB}}]}{(N-1)!} \Bigg) \\ - \lambda A_S e^{-\lambda t A_S} \cdot \Bigg(N! \sum_{i=M}^{N-2} \left(\frac{e^{-\lambda t A_{LB} i} [1 - e^{-\lambda t A_{LB}}]^{N-i}}{i!(N-i)!} \right) \\ + \frac{N! e^{-\lambda t A_{LB} i} [1 - e^{-\lambda t A_{LB}}]}{(N-1)!} + e^{-\lambda t A_{LB} N} \Bigg) \end{aligned} \tag{5.16}$$

Nach dem Einsetzen von $t = 0$ ergibt sich der Anstieg von $R_{RLB}(0)$:

$$R'_{RLB}(0) = -A_S \cdot \lambda \tag{5.17}$$

Somit kann ein positiver Effekt für die Zuverlässigkeit erreicht werden, solange das Verhältnis der Flächen $\frac{A_S}{A_O} < 1$ ist. Darüber hinaus wird ersichtlich, dass die Zuverlässigkeit $R_{M \text{ aus } N}(0) = 1$ und die 1. Ableitung von $R'_{M \text{ aus } N}(0) = 0$ ist.

Grundsätzlich wurde mit der vorgestellten Modellierung die Möglichkeit geschaffen, eine beliebige Anzahl identischer Prozessorkomponenten dahingehend zu bewerten, ob zusätzliche Redundanz mit der vorgestellten Architektur dieser Arbeit überhaupt einen positiven Effekt auf die resultierende Zuverlässigkeit haben kann.

In der vorgestellten Modellierung der Zuverlässigkeit des System und der damit verbundenen Bewertung der Effizienz einer Redundanzstrategie wurde bis hier davon ausgegangen, dass der gewählte Test der Logik-Blöcke in der Lage ist alle Fehler zu erkennen. Allerdings ist davon auszugehen, dass nicht alle Fehler eines Fehlermodells durch den Test erkannt werden. Üblicherweise wird für jeden Test auf Fehler eines jeden Fehlermodells eine Fehlerüberdeckung (engl. fault coverage (FC)) angegeben. Diese Fehlerüberdeckung bezeichnet den prozentualen Anteil aller Fehler eines Fehlermodells, welche durch den gewählten Test erkannt werden. Werden alle Fehler durch den Test erkannt, entspricht die Bewertung der Effizienz einer Redundanzstrategie der eben vorgestellten Modellierung. Damit auch eine Bewertung der Redundanzstrategie bei Testverfahren, welche keine vollständige Fehlerüberdeckung leisten, möglich ist, wird das vorgestellte Modell wie folgt erweitert.

Die Fehlerüberdeckung eines LB teilt jetzt seine Fläche A_{LB} in die Flächen $A_{LB_{FC}}$ und $A_{LB_{NFC}}$. Hier steht $A_{LB_{FC}}$ für den Teil des LB, in welchem Fehler durch den Test erkannt werden und $A_{LB_{NFC}}$ für den Teil des LB in welchem Fehler nicht erkannt werden. Dabei wird hier angenommen, dass alle Fehler des jeweiligen Fehlermodells über die gesamte Fläche eines Logik-Blocks gleichverteilt sind.

$$A_{LB} = A_{LB_{FC}} + A_{LB_{NFC}} \tag{5.18}$$

In Abbildung 5.3 ist die Aufteilung der Flächen des Ausgangssystems und ebenfalls der Flächen des Systems mit Redundanz unter Berücksichtigung einer Fehlerüberdeckung dargestellt.

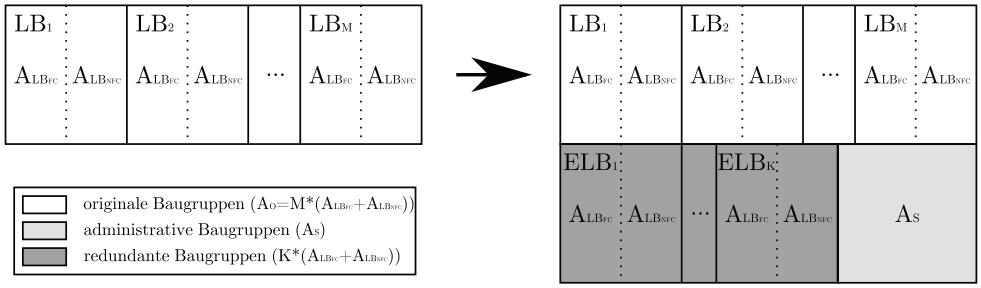


Abbildung 5.3: Flächenaufteilung des originalen (*links*) und des modifizierten Systems (*rechts*) unter Einbeziehung der Fehlerüberdeckung der LB/ELB

Die Zuverlässigkeit $R_{LB}(t)$ eines LB ergibt sich dann nach folgender Serienkomposition:

$$R_{LB}(t) = R_{LB_{FC}}(t) \cdot R_{LB_{NFC}}(t) \tag{5.19}$$

Die Zuverlässigkeit $R_{RLB}(t)$ des resultierenden RLB kann dann durch Gleichung 5.20 errechnet werden.

$$R_{RLB}(t) = R_S(t) \cdot R_{LB_{NFC}}(t)^N \cdot \sum_{i=M}^{N} \binom{N}{i} R_{LB_{FC}}(t)^i \cdot [1 - R_{LB_{FC}}(t)]^{N-i} \tag{5.20}$$

Durch Umstellen erhält man die Zuverlässigkeit $R_{LB_{FC}}(t)$ der Fläche des LB in welchem Fehler erkannt werden:

$$R_{LB_{FC}}(t) = R_a(t)^{A_{LB_{FC}}} \tag{5.21}$$

In gleicher Weise lässt sich die Zuverlässigkeit $R_{LB_{NFC}}(t)$ der Fläche des LB, in welchem der Test nicht in der Lage ist Fehler zu erkennen, folgend modellieren:

$$R_{LB_{NFC}}(t) = R_a(t)^{A_{LB_{NFC}}} \tag{5.22}$$

Durch Einsetzen der Gleichungen 5.21 und 5.22 in Gleichung 5.20 erhält man:

$$R_{RLB}(t) = R_a(t)^{A_S} \cdot R_a(t)^{N \cdot A_{LB_{NFC}}} \sum_{i=M}^{N} \binom{N}{i} \cdot R_a(t)^{i \cdot A_{LB_{FC}}} \cdot [1 - R_a(t)^{A_{LB_{FC}}}]^{N-i} \tag{5.23}$$

Nach Vereinfachung mit:

$$A_{LB_{FC}} = A_{LB} \cdot FC \tag{5.24}$$

$$A_{LB_{NFC}} = A_{LB} \cdot (1 - FC) \tag{5.25}$$

erhält man:

$$R_{RLB}(t) = R_a(t)^{A_S} \cdot R_a(t)^{N \cdot A_{LB} \cdot (1-FC)} \sum_{i=M}^{N} \binom{N}{i} R_a(t)^{i \cdot A_{LB} \cdot FC} \cdot [1 - R_a(t)^{A_{LB} \cdot FC}]^{N-i} \tag{5.26}$$

Um jetzt zu bewerten, ob der Einsatz von Fehlertoleranz für eine beliebige Anzahl von Prozessorkomponenten bei einer gegebenen Fehlerüberdeckung zu einer Steigerung der Zuverlässigkeit führen kann, interessiert wieder der Anstieg der Funktion zum Zeitpunkt $t = 0$. Sobald $R'_{RLB}(0) > R'_O(0)$ kommt es in jedem Fall zu einer Steigerung der Zuverlässigkeit für $t \to 0$.

Vereinfachung der Gleichung 5.26 durch Substitution ergibt:

$$u(t) = R_a(t)^{A_S} \tag{5.27}$$

$$v(t) = R_a(t)^{N \cdot A_{LB} \cdot (1-FC)} \tag{5.28}$$

$$w(t) = \sum_{i=M}^{N} \binom{N}{i} R_a(t)^{i \cdot A_{LB} \cdot FC} \cdot [1 - R_a(t)^{A_{LB} \cdot FC}]^{N-i} \tag{5.29}$$

$$R_{RLB}(t) = u(t)v(t)w(t) \tag{5.30}$$

Nach Bildung der Ableitung von $\frac{dR_{RLB}(t)}{dt}$ erhält man:

$$R'_{RLB}(t) = u'(t)v(t)w(t) + u(t)v'(t)w(t) + u(t)v(t)w'(t) \tag{5.31}$$

Setzt man nun für $t = 0$ ein vereinfacht sich die Gleichung, da $u(0) = v(0) = w(0) = 1$ sind:

$$R'_{RLB}(0) = u'(0) + v'(0) + w'(0) \tag{5.32}$$

Bereits in Gleichung 5.16 wurde gezeigt, dass $w'(0) = 0$ ist und somit folgt:

$$R'_{RLB}(0) = u'(0) + v'(0) \tag{5.33}$$

Damit ergibt sich der Anstieg zum Zeitpunkt $t = 0$ durch:

$$R'_{RLB}(0) = -A_S\lambda - A_{LB}(1 - FC)N\lambda \tag{5.34}$$

Für den Fall, dass die Fehlerüberdeckung nicht 100 Prozent beträgt, wird die Redundanzstrategie mit zunehmender Anzahl von ELBs weniger effizient. Diese Entwicklung ist in Abbildung 5.4 verdeutlicht. Hier ist die resultierende Zuverlässigkeit $R_{RLB}(t)$ für ein System mit einem originalen LB, einer konstanten Fehlerüberdeckung FC von 90% und einem angenommenen konstanten Verhältnis von $\frac{A_S}{A_O} = \frac{1}{10}$ für $2, 4, 6$ und 8 ELBs dargestellt.

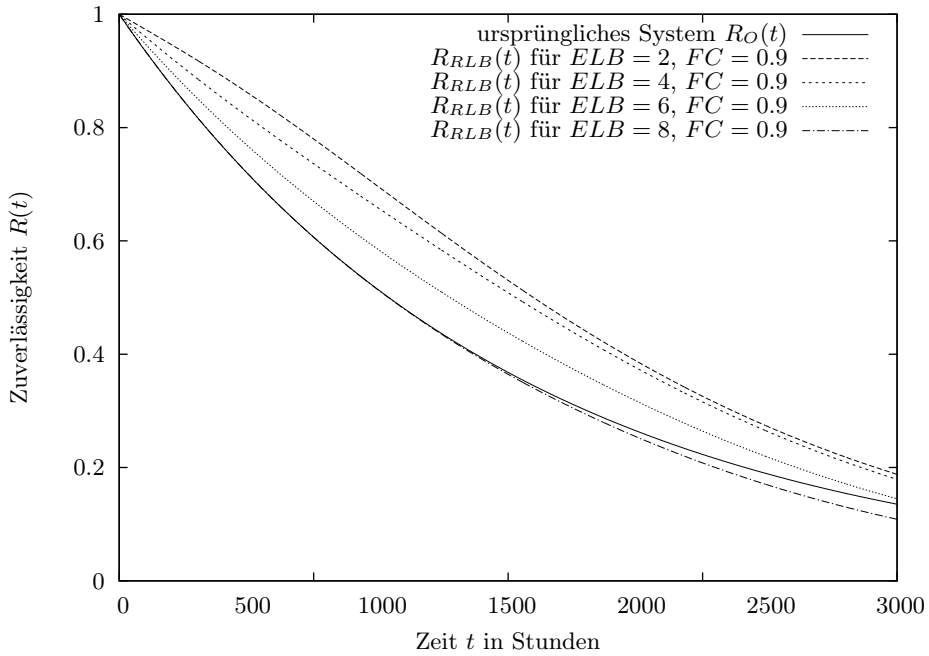


Abbildung 5.4: Entwicklung der Zuverlässigkeit $R_O(t)$ und $R_{RLB}(t)$ für identische Flächenverhältnisse $r = \frac{A_S}{A_O} = \frac{1}{10}$ für ein System mit $M = 1$, $FC = 0.9$, $A_O = 10^6$ bei steigender Anzahl von ELBs mit $\lambda_a = 10^{-9}$

Eine Steigerung der Zuverlässigkeit durch den Einsatz von ELBs in einem RLB kann, durch die eben vorgestellte in Gleichung 5.26 resultierende Erweiterung, jetzt auch in Bezug auf die Fehlerüberdeckung des verwendeten Tests, bewertet werden.

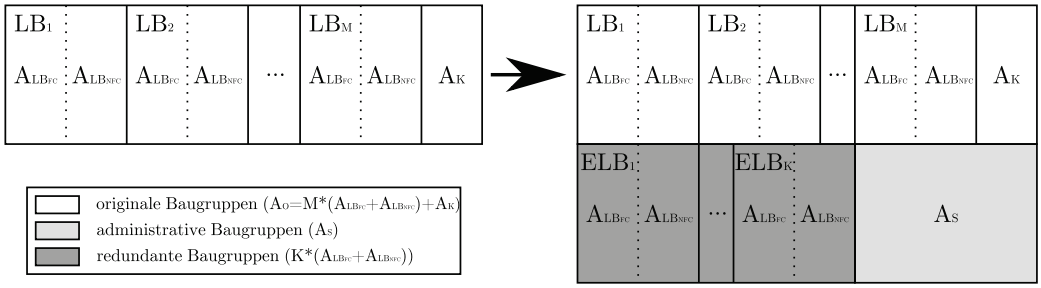


Abbildung 5.5: Flächenaufteilung des originalen und des modifizierten Systems mit der zusätzlichen Fläche A_K im ursprünglichen System

Die Annahme im vorgestellten Modell, dass sich die Fläche des Ausgangssystems A_O aus genau M identischen Teilen zusammensetzt, diente der Bewertung einer spezifischen Redundanzstrategie für einen RLB. Folgend soll gezeigt werden, wie die Berechnung der Zuverlässigkeit eines Gesamtsystems in dieser Arbeit erfolgt.

In Abbildung 5.5 ist die Aufteilung des originalen und des modifizierten Systems erweitert um die Teilfläche A_K, welche keine Redundanz zugeteilt bekommt, dargestellt. Damit lassen sich die Zuverlässigkeiten beider Systeme wie folgt beschreiben:

$$R_O(t) = R_K(t) \cdot R_{LB}(t)^M \tag{5.35}$$

$$R_{System}(t) = R_K(t) \cdot R_S(t) \cdot R_{M \text{ aus } N}(t) \tag{5.36}$$

In beiden Modellierungen ist die Zuverlässigkeit der Fläche A_K ein weiterer Faktor im Seriensystem. Dieser beeinflusst zwar die resultierende Zuverlässigkeit des Systems und damit die Effizienz der Redundanzstrategie auf das Gesamtsystem, aber nicht, ob eine bestimmte Konfiguration überhaupt eine Zuverlässigkeitssteigerung bewirken kann.

Der Einfluss einer Segmentierung kann ebenfalls durch die vorgestellte Herangehensweise modelliert werden. In Abbildung 5.6 wird verdeutlicht wie die Aufteilung der Flächen bei einer Segmentierung ausfällt.

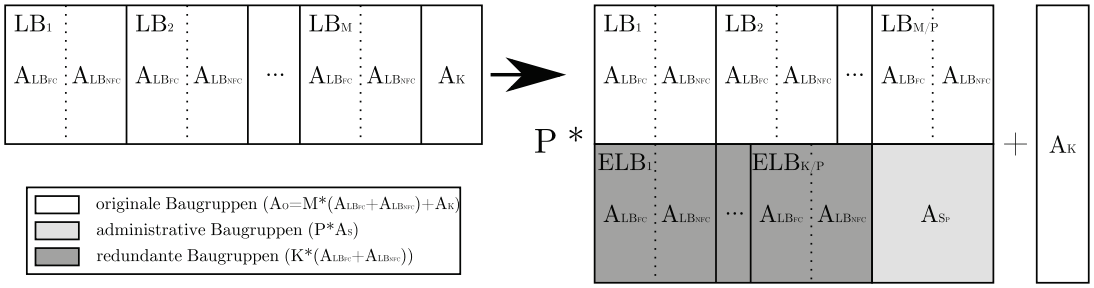


Abbildung 5.6: Flächeneinteilung bei Segmentierung

Hierzu wird das Ausgangssystem mit M identischen Logik-Blöcken in P Subsysteme mit dedizierter Redundanz und administrativen Baugruppen und einer gemeinsamen nicht mit Redundanz versehenen Fläche A_K eingeteilt. Die Zuverlässigkeit des Gesamtsystems ist dann folgendermaßen zu beschrieben:

$$R_{RLB_P}(t) = R_{S_P}(t) \cdot R_{M \text{ aus } N_P}(t) \tag{5.37}$$

$$R_{System}(t) = R_K(t) \cdot \prod_{i=1}^{P} R_{RLB_P}(t) \tag{5.38}$$

Die Effizienz eines Segmentes ist somit durch die Berechnung von $R_{RLB_P}(t)$ zu bestimmen und entspricht der bereits vorgestellten Herangehensweise.

Wird das ursprüngliche Design in L RLBs eingeteilt, dann ist die resultierende Zuverlässigkeit durch folgende Gleichung zu ermitteln:

$$R_{System}(t) = R_K(t) \cdot \prod_{i=1}^{L} R_{RLB_i}(t) \tag{5.39}$$

Für den Vergleich verschiedener RLB-Konfigurationen in Bezug auf eine Zuverlässigkeitssteigerung wird in dieser Arbeit der RIF (*reliability improvement factor* [Lal01]) verwendet. Dieser beschreibt die Steigerung der Zuverlässigkeit zu einem beliebigen Zeitpunkt T.

$$RIF = \frac{1 - R_O}{1 - R_{System}} \tag{5.40}$$

Die Bestimmung der optimalen RLB-Konfiguration für ein Gesamtsystem kann dann in folgenden Teilschritten erfolgen:

1. Festlegen einer zu erreichenden Zuverlässigkeit R_{System} des Gesamtsystems
2. Synthese des Ausgangssystems zur Ermittlung der:
   - Gesamtfläche des Designs A_O
   - Fläche A_{LB} und Anzahl M von Logik-Blöcken einschließlich der Anzahl ihrer Ein- und Ausgänge I_{LB} und O_{LB}
3. Auswahl des Testverfahrens und Ermittlung der Fehlerüberdeckung FC der jeweiligen Logik-Blöcke
4. Ermittlung der administrativen Fläche A_S der jeweiligen RLB durch das Variieren der Anzahl der Ersatz-Logik-Blöcke K und der Einteilung eines RLB in verschiedene Segmente P
5. Lösen der Gleichung für $R_O(t)$ und $R_{System}(t)$ zur zielorientierten Optimierung der RLB-Konfiguration unter Berücksichtigung eines maximalen Hardware-Mehraufwandes

Die Entscheidung für eine RLB-Konfiguration kann durch die Festlegung einer Anforderung an die Zuverlässigkeit und einer Begrenzung der zusätzlich benötigten Fläche erreicht werden. Dafür kann beispielsweise die resultierende Zuverlässigkeit R_{System} festgelegt werden, welche erreicht werden soll. Durch Variieren der RLB-Konfiguration also der Parameter K und P kann eine optimale Konfiguration für den jeweiligen RLB gefunden werden. Die Möglichkeiten der Variation des Parameters P ist begrenzt durch die Anzahl der zur Verfügung stehenden LBs. Ein maximal segmentierter RLB kann genau M Segmente beinhalten, was separate ELBs für jeden LB bedeutet. Der Parameter K kann prinzipiell beliebig erhöht werden, allerdings erhöht sich damit zum ersten der notwendige Hardware-Mehraufwand und zum anderen auch die zusätzliche administrative Fläche A_S. Wie bereits gezeigt, ist eine Steigerung der Zuverlässigkeit für Flächenverhältnisse $\frac{A_S}{A_O} > 1$ ausgeschlossen.

Die vorgestellte Modellierung der Zuverlässigkeit und die daraus resultierenden Ergebnisse werden im folgenden an zwei Beispielen verdeutlicht. Für diese Beispiele wurden die Flächen für den RLB-Controller und den RLB-Speicher vernachlässigt.

Beispiel 1 sei System A bestehend aus:

- $M = 4$ (Anzahl identischer LB)
- $A_{LB} = 2000$ (Fläche eines LB)
- $I = 40$ (Anzahl der Eingänge eines LB)
- $O = 20$ (Anzahl der Ausgänge eines LB)

Tabelle 5.1 beinhaltet die Ergebnisse für verschiedene RLB-Konfigurationen. Die erste Spalte beinhaltet die jeweilige Konfiguration des RLB mit Art der Segmentierung P und Anzahl $\frac{M}{P}$ der ursprünglichen LBs sowie die Anzahl $\frac{K}{P}$ der zusätzlichen ELB für jedes Segment. Der resultierende Hardware Mehraufwand der jeweiligen RLB-Konfiguration steht in Spalte 2. In den folgenden Spalten findet sich der RIF der jeweiligen RLB-Konfiguration und dem ursprünglichen System bei einer zu erreichenden Zuverlässigkeit von $R_{RLB} = 0.9|0.99|0.999$. In den Spalten 3, 5 und 7 wird von einer erreichten Fehlerüberdeckung $FC = 100\%$ und in den restlichen Spalten von einer Fehlerüberdeckung von $FC = 95\%$ ausgegangen.

Zuverlässigkeitsteigerung Beispiel 1 (System A)							
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	RIF **für** $R_{RLB} = 0.9$		RIF **für** $R_{RLB} = 0.99$		RIF **für** $R_{RLB} = 0.999$	
		FC 100%	FC 95%	FC 100%	FC 95%	FC 100%	FC 95%
1x(4+1)	32.5	3.26	2.98	7.84	**5.75** [1-3]	**12.00** [1-3]	**7.00** [1-3]
1x(4+2)	61	4.51 [1]	**3.75** [1-3]	8.56	5.27	9.00	5.40
1x(4+3)	89.5	4.75 [2]	3.57	6.69	4.23	6.90	4.30
1x(4+4)	118	4.38	3.13	5.43	3.53	5.50	3.60
2x(2+1)	58	3.72	3.23	8.56 [1,2]	5.61	11.80	6.30
2x(2+2)	112	**4.86** [3]	3.59	7.98	4.46	8.30	4.50
2x(2+3)	166	4.69	3.08	6.09	3.46	6.20	3.50
4x(1+1)	109	4.03	3.23	**8.59** [3]	4.87	10.70	5.20
4x(1+2)	214	4.80	3.00	6.91	3.41	7.10	3.40
4x(1+3)	319	4.22	2.37	5.15	2.54	6.20	2.60

[Zahl] - Optimale RLB-Konfiguration (keine Beschränkung der zusätzlich notwendigen Hardware)
[1] Optimale RLB-Konfiguration bei maximal 75% HW-Mehraufwand
[2] Optimale RLB-Konfiguration bei maximal 100% HW-Mehraufwand
[3] Optimale RLB-Konfiguration bei maximal 150% HW-Mehraufwand

Tabelle 5.1: Einfluss der verschiedenen RLB-Konfigurationen auf die Zuverlässigkeit für Beispiel 1 (System A)

Ein größerer RIF bedeutet einen höheren Gewinn im Sinne der Steigerung der Zuverlässigkeit zwischen dem ursprünglichen System und der jeweiligen RLB-Konfiguration. Die Konfiguration mit dem höchsten Wert für den RIF jeder Spalte ist **fett** dargestellt und beschreibt die optimale Konfiguration für das gegebene System bei der jeweilig zu erreichenden Zuverlässigkeit. Die Fußnoten 1-3 zeigen die beste RLB-Konfiguration für eine definierte Obergrenze an zusätzlicher Hardware von 75%, 100% beziehungsweise 150%.

Als zweites Beispiel dient ein System bestehend aus dem betrachteten System A und einem System B mit folgenden Eigenschaften:

- $M = 6$ (Anzahl identischer LB)
- $A_{LB} = 5000$ (Fläche eines LB)
- $I = 40$ (Anzahl der Eingänge eines LB)
- $O = 20$ (Anzahl der Ausgänge eines LB)

Die Fehlerüberdeckung FC der Logik-Blöcke von System A sei 100% und die von System B sei 97%. Tabelle 5.2 beinhaltet die Ergebnisse für Beispiel 2. In der linken Tabellenhälfte sind die jeweiligen Konfigurationen der RLBs für System A und B mit dem resultierenden Hardware Mehraufwand und dem resultierenden RIF für eine zu erreichende Zuverlässigkeit $R_{System} = 0.9$ dargestellt. Die rechte Tabellenhälfte beinhaltet die Ergebnisse für $R_{System} = 0.99$. Aufgrund der Vielzahl der möglichen Kombinationen von RLB-Konfigurationen sind hier die 10 Kombinationen mit der größten Zuverlässigkeitssteigerung in absteigender Reihenfolge aufgelistet.

Zuverlässigkeitsteigerung Beispiel 2 (System A + System B)							
RLB-Konfigurationen		**extra Hard-ware (%)**	RIF **für** $R_{System} = 0.9$	**RLB-Konfigurationen**		**extra Hard-ware (%)**	RIF **für** $R_{System} = 0.99$
System A	System B			System A	System B		
1x(4+2)	1x(6+3)	56.74	5.93	2x(2+1)	1x(6+2)	41.89	11.09
2x(2+2)	1x(6+3)	67.47	5.90	1x(4+1)	1x(6+2)	36.53	11.06
4x(1+1)	1x(6+3)	66.84	5.85	4x(1+1)	1x(6+2)	52.63	10.92
4x(1+2)	1x(6+3)	88.95	5.81	1x(4+2)	1x(6+2)	42.53	10.59
2x(2+1)	1x(6+3)	56.11	5.80	2x(2+2)	1x(6+2)	53.26	10.37
1x(4+2)	1x(6+4)	70.95	5.80	2x(2+1)	2x(3+2)	68.42	10.11
1x(4+3)	1x(6+3)	62.74	5.79	1x(4+1)	2x(3+2)	63.05	10.10
2x(2+2)	1x(6+4)	81.68	5.77	2x(2+1)	2x(3+1)	40.95	10.09
1x(4+2)	2x(3+2)	69.05	5.77	1x(4+1)	2x(3+1)	35.58	10.08
1x(4+2)	2x(3+3)	96.53	5.75	2x(2+1)	3x(2+1)	54.21	10.08

Tabelle 5.2: Einfluss der verschiedenen RLB-Konfigurationen auf die Zuverlässigkeit für Beispiel 2 (System A + System B)

5.1.2 Modellierung der mittleren Lebensdauer

Wie bereits in Abschnitt 2.2.2 beschrieben, ergibt sich die mittlere Zeit bis zum Ausfall eines Systems aus der Integration der Zuverlässigkeit über die Zeit im Intervall von 0 bis ∞. Die Modellierung der Funktion für die Zuverlässigkeit des modifizierten Systems $R_{System}(t)$ wurde im letzten Abschnitt ausführlich behandelt. Durch die Einteilung des Systems in Einheitsflächen mit identischer Zuverlässigkeit $R_a(t)$ wurde die resultierende Zuverlässigkeit des ursprünglichen Systems $R_O(t)$ und die des modifizierten Systems $R_{System}(t)$ anhand ihrer Fläche beschrieben.

Bereits in [5] und [14] wurde für den Vergleich der MTTF zweier Systeme der LIF (lifetime improvement factor) verwendet. Dieser beschreibt das Verhältnis zwischen der mittleren Lebensdauer von modifizierten und ursprünglichen System:

$$LIF = \frac{MTTF_{System}}{MTTF_O} \tag{5.41}$$

Zur Ermittlung des LIF sind die Funktionen für die Zuverlässigkeit beider Systeme zu integrieren. Aufgrund der Komplexität der Modellierung der Zuverlässigkeit für das modifizierte System $R_{System}(t)$, wurde die Integration hier numerisch gelöst.

Tabelle 5.3 zeigt den LIF für das im vorherigen Abschnitt verwendete System A. In den Spalten 1 und 2 sind wieder die jeweilige RLB-Konfiguration und der resultierende Hardware-Mehraufwand aufgelistet. Die Spalten 3-5 beinhalten den LIF für angenommene Fehlerüberdeckungen von 100%, 95% und 90% der Logik-Blöcke.

Lebensdauersteigerung Beispiel 1 (System A)				
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	LIF (*lifetime improvement factor*)		
		FC = 100%	**FC = 95%**	**FC = 90%**
1x(4+1)	32.5	1.63204	1.57865	1.52647
1x(4+2)	61	2.07672	1.94009	1.81332
1x(4+3)	89.5	2.35003[1]	2.12055[1]	**1.91856**[1-3]
1x(4+4)	118	2.49636	2.17855	1.91300
1x(4+5)	146.5	2.55112	2.15772	1.84515
1x(4+6)	175	2.54187	2.08925	1.74637
2x(2+1)	58	1.84174	1.73745	1.63990
2x(2+2)	112	2.37279	2.09952	1.86735
2x(2+3)	166	2.64372[2]	**2.19495**[2-3]	1.84882
2x(2+4)	220	2.73946[3]	2.14299	1.72370
2x(2+5)	274	2.72356	2.02020	1.56642
2x(2+6)	328	2.64069	1.87107	1.41132
4x(1+1)	109	2.02201	1.82465	1.65340
4x(1+2)	214	2.56855	2.06363	1.69537
4x(1+3)	319	**2.76258**	1.98732	1.50868
4x(1+4)	424	2.75195	1.79916	1.29195
4x(1+5)	529	2.63415	1.59390	1.10496
4x(1+6)	634	2.46817	1.40730	0.95586

[Zahl] - Optimale RLB-Konfiguration (keine Beschränkung der zusätzlich notwendigen Hardware)
[1] Optimale RLB-Konfiguration bie maximal 100% HW-Mehraufwand
[2] Optimale RLB-Konfiguration bie maximal 200% HW-Mehraufwand
[3] Optimale RLB-Konfiguration bie maximal 300% HW-Mehraufwand

Tabelle 5.3: Einfluss der verschiedenen RLB-Konfigurationen auf den LIF für Beispiel 1 (System A)

Die optimale RLB-Konfiguration ist wieder durch die **fett** dargestellte Zahl jeder Spalte hervorgehoben. Die Fußnoten 1-3 entsprechen hier der optimalen RLB-Konfiguration für eine definierte Obergrenze an zusätzlicher Hardware von 100%, 200% und 300%.

In Tabelle 5.4 ist die Steigerung der Lebensdauer in Form des LIF für Beispiel 2 (System A + System B) aufgelistet. Die linke Seite der Tabelle beinhaltet die jeweiligen RLB-Konfigurationen von System A und System B mit dem dazugehörigen Hardware-Mehraufwand und dem LIF für einen maximal zulässigen

Lebensdauersteigerung Beispiel 2 (System A + System B)							
RLB-Konfigurationen		**extra Hard-ware (%)**	LIF	**RLB-Konfigurationen**		**extra Hard-ware (%)**	LIF
System A	System B			System A	System B		
1x(4+2)	1x(6+6)	99.37	3.7286	2x(2+3)	2x(3+5)	173.58	3.9655
2x(2+2)	1x(6+5)	95.89	3.7140	4x(1+2)	2x(3+5)	183.68	3.9619
1x(4+3)	1x(6+5)	91.16	3.7034	2x(2+2)	2x(3+5)	162.21	3.9414
1x(4+4)	1x(6+5)	97.16	3.6854	1x(4+3)	2x(3+5)	157.47	3.9316
1x(4+2)	1x(6+5)	85.16	3.6376	1x(4+4)	2x(3+5)	163.47	3.9212
1x(4+2)	2x(3+3)	96.53	3.5672	2x(2+2)	2x(3+6)	189.68	3.9121
4x(1+1)	1x(6+5)	95.26	3.5644	2x(2+3)	3x(2+4)	199.16	3.9089
2x(2+1)	1x(6+6)	98.74	3.5595	2x(2+4)	2x(3+5)	184.95	3.9087
2x(2+3)	1x(6+4)	93.05	3.5250	1x(4+3)	2x(3+6)	184.95	3.9028

Tabelle 5.4: Einfluss der verschiedenen RLB-Konfigurationen auf den LIF für Beispiel 2 (System A + System B)

Hardware-Mehraufwand von 100%. Auf der rechten Seite finden sich die Ergebnisse für einen Mehraufwand an Hardware von maximal 200%.

5.1.3 Modellierung der Produktionsausbeute

In den vorangegangenen beiden Abschnitten wurde gezeigt, wie sich die Zuverlässigkeit und daraus resultierend die mittlere Lebensdauer ermitteln lässt. Gleichzeitig konnte aus der gewählten Modellierung abgeleitet werden, welche RLB-Konfiguration für ein spezifisches System den größten Vorteil zum Erreichen einer bestimmten Zuverlässigkeit beziehungsweise mittleren Lebensdauer bedeutet. Jede RLB-Konfiguration hat Auswirkungen auf die Produktionsausbeute, da zusätzliche Fläche für Redundanz und deren Administration benötigt wird und daher weniger Chips auf einem Wafer platziert werden können. Die Anzahl der Chips pro Wafer wird durch den Durchmesser des Wafers und die Fläche eines Chips bestimmt.

Die in dieser Arbeit vorgestellte Architektur ist so ausgelegt, dass auch Produktionsfehler in redundanten Baugruppen kompensiert werden können. Die Kompensation von Produktionsfehlern führt dann allerdings zu einem Verlust der Möglichkeit der Fehlertoleranz im Feld. Diese Eigenschaft kann allerdings auch eingesetzt werden, um die Produktionsausbeute zu steigern. Insgesamt werden bei einer Architektur mit Redundanz zwar weniger Chips pro Wafer produziert,

diese können aber trotz Produktionsfehlern noch funktionstüchtig sein, wenn der Fehler nur redundante Teile des Chips betrifft.

Die Modellierung des Systems ist equivalent zur Modellierung der Zuverlässigkeit aus Abschnitt 5.1.1. Lediglich in der Verteilungsfunktion für ein Flächenequivalent ist die Fehlerrate durch eine Fehlerwahrscheinlichkeit auszutauschen, da die Dimension der Zeit für Fertigungsfehler nicht relevant ist.

Für eine gegebene Fehlerwahrscheinlichkeit pro Fläche ist die Wahrscheinlichkeit P_O eines funktionsfähigen Chips ohne Redundanz zu ermitteln. Ebenso kann die Wahrscheinlichkeit P_{System} für funktionsfähige Chips mit Redundanz ermittelt werden. Dabei gilt der Chip als funktionsfähig solange Fehler in Komponenten durch Redundanz kompensiert werden können. Die resultierenden Wahrscheinlichkeiten für das ursprüngliche und modifizierte Design multipliziert mit der jeweiligen Anzahl von CPW (Chips pro Wafer) ergibt dann die $FCPW$ (funktionsfähige Chips pro Wafer).

$$FCPW_O = CPW_O \cdot P_O \tag{5.42}$$

$$FCPW_{System} = CPW_{System} \cdot P_{System} \tag{5.43}$$

Anschließend lässt sich wie bei der Steigerung der mittleren Lebensdauer ein Verhältnis bestimmen. Dieses wird für die Steigerung der Produktionsausbeute folgend als YIF (*yield improvement factor*) bezeichnet.

$$YIF = \frac{FCPW_{System}}{FCPW_O} \tag{5.44}$$

In Tabelle 5.5 sind die Verhältnisse für verschiedene Produktionsausbeuten für das in den vorhergehenden Abschnitten verwendete System A angegeben. Spalten 1 und 2 beinhalten die jeweilige RLB-Konfiguration sowie den Hardware-Mehraufwand. Die Spalten 3-6 zeigen das Verhältnis der funktionierenden Chips von modifizierten und ursprünglichen System für ursprüngliche Ausbeuten von 25%, 50%, 75% und 95%. Eine Steigerung der Produktionsausbeute wird für $YIF > 1$ erreicht.

Ausbeutesteigerung Beispiel 1 (System A)					
RLB-Konfiguration P**x**$(\frac{M}{P}+\frac{K}{P})$	**extra Hardware (%)**	YIF **für verschiedene Ausbeuten**			
		25%	**50%**	**75%**	**95%**
1x(4+1)	32.5	1.477	**1.172**	0.943	0.826
1x(4+2)	61	**1.615**	1.087	0.798	0.682
1x(4+3)	89.5	1.524	0.940	0.674	0.577
1x(4+4)	118	1.354	0.806	0.580	0.500
2x(2+1)	58	1.424	1.040	0.802	0.694
2x(2+2)	112	1.356	0.843	0.606	0.517
2x(2+3)	166	1.137	0.669	0.478	0.410
4x(1+1)	109	1.179	0.811	0.609	0.525
4x(1+2)	214	0.947	0.568	0.507	0.348

[Zahl] - Optimale RLB-Konfiguration

Tabelle 5.5: Einfluss der verschiedenen RLB-Konfigurationen auf den YIF für Beispiel 1 (System A)

Ausbeutesteigerung Beispiel 2 (System A + System B)							
RLB-Konfigurationen		**extra Hardware (%)**	YIF **bei 50%**	**RLB-Konfigurationen**		**extra Hardware (%)**	YIF **bei 75%**
System A	System B			System A	System B		
1x(4+1)	1x(6+1)	22.32	1.3670	1x(4+1)	1x(6+1)	22.32	1.0400
1x(4+1)	1x(6+2)	36.53	1.3340	2x(2+1)	1x(6+1)	27.68	0.9970
2x(2+1)	1x(6+1)	27.68	1.3150	1x(4+2)	1x(6+1)	28.32	0.9920
1x(4+2)	1x(6+1)	28.32	1.3110	1x(4+1)	1x(6+2)	36.53	0.9490
2x(2+1)	1x(6+2)	41.89	1.2880	1x(4+1)	2x(3+1)	35.58	0.9460
1x(4+2)	1x(6+2)	42.53	1.2850	1x(4+3)	1x(6+1)	34.32	0.9450
1x(4+1)	2x(3+1)	35.58	1.2820	4x(1+1)	1x(6+1)	38.42	0.9190
1x(4+3)	1x(6+1)	34.32	1.2470	2x(2+2)	1x(6+1)	39.05	0.9140
2x(2+1)	2x(3+1)	40.95	1.2370	2x(2+1)	1x(6+2)	41.89	0.9130

Tabelle 5.6: Einfluss der verschiedenen RLB-Konfigurationen auf den YIF für Beispiel 2 (System A + System B)

In Tabelle 5.6 ist der Einfluss der RLB-Konfiguration auf die Produktionsausbeute für Beispiel 2, bestehend aus System A und System B, dargestellt. Die linke Tabellenhälfte zeigt das Verhältnis von funktionierenden Chips pro Wafer bei einer ursprünglichen Produktionsausbeute pro Chip von 50% und die rechte Hälfte für eine Produktionsausbeute pro Chip von 75%.

5.2 Grenzen der Modellierung

In diesem Kapitel wurde dargestellt, wie Zuverlässigkeit, mittlere Lebensdauer und Produktionsausbeute in dieser Arbeit modelliert wurden. Die vorgestellte Modellierung ermöglicht einen gezielten und schnellen Auswahlprozess für die vorgestellte aktive Hardware-Redundanz zur Kompensation von permanenten Fehlern dieser Arbeit. An dieser Stelle sollen Grenzen und Randbedingungen der gewählten Modellierung aufgezeigt werden.

Damit eine Evaluation verschiedener RLB-Konfigurationen stattfinden kann, sind zuerst alle notwendigen Parameter zu ermitteln. Da die administrative Fläche A_S stets von der verwendeten RLB-Konfiguration abhängt, zieht die Ermittlung dieser Fläche für jede Konfiguration eine Logik-Synthese mit anschließendem Layout nach sich. Allerdings wurde bereits in [5] gezeigt, dass die zusätzliche Fläche auch durch Approximieren ermittelt werden kann.

Die Annahme einer identischen Verteilungsfunktion zur Modellierung von Zuverlässigkeit, mittlerer Lebensdauer und Produktionsausbeute stellt hier eine wesentliche Vereinfachung dar. Denn hier wird jeder Fläche des ICs eine gleiche Anfälligkeit gegenüber permanenten Fehlern unterstellt, welche in der Realität nicht gegeben sein muss. Gerade für Alterungseffekte (siehe Abschnitt 2.3.2) ist die Anfälligkeit besonders von der Temperatur abhängig, welche bei unterschiedlichen Anwendungsszenarien stark variieren kann. Ebenfalls wird damit der Einfluss verschiedener Fehlereffekte auf unterschiedliche Strukturen des ICs vernachlässigt. Gerade für die zusätzlichen redundanten Baugruppen bedeutet diese Modellierung eine Ungenauigkeit, weil hier angenommen wird, dass auch nicht verwendete Komponenten des ICs eine identische Anfälligkeit gegenüber Fehlereffekten besitzen wie verwendete. Aus diesem Grund ist die Modellierung und die damit gewonnenen Ergebnisse als ein *Worst-Case*-Szenario zu betrachten. Jedoch ermöglicht diese Modellierung eine Evaluierung verschiedener Redundanzstrategien bereits im Entwurfsprozess.

Stellt sich eine bestimmte Fehlerursache für eine spezielle Technologie oder einen speziellen Anwendungsfall als besonders relevant heraus, ist es möglich, dies in der gewählten Modellierung zu berücksichtigen. Beispielsweise kann eine unterschiedliche Anfälligkeit bestimmter Komponenten eines Prozessors durch die Verwendung verschiedener Fehlerraten modelliert werden.

KAPITEL 6

Anwendungsbeispiel

Die Umsetzbarkeit des vorgestellten Verfahrens dieser Arbeit wird im folgenden an einer Beispielarchitektur gezeigt. Dafür wird die verwendete Prozessorarchitektur mit dem dazugehörigen Befehlssatz vorgestellt. Anschließend werden alle Modifikationen und Erweiterungen der Architektur erläutert. Ebenfalls wird auf den Entwurf und die Entwicklung der verwendeten Test-Software eingegangen. Abschließend werden die erzielten Ergebnisse der Entwurfsraumexploration für eine Konfiguration der verwendeten Prozessorarchitektur dargestellt und diskutiert.

Bei der verwendeten Architektur handelt es sich um einen *very long instruction word* (VLIW) Prozessor. Laut der Klassifizierung von Prozessoren nach Flynn [Fly72] handelt es sich dabei um eine *single instruction multiple data* (SIMD) Architektur. Diese Klasse von Prozessoren besitzt einen Befehlsstrom und typischerweise mehrere Datenströme. Der Befehlsstrom selbst besteht bei einer solchen Architektur aus mehreren Befehlen, die in einem langen Instruktionswort zusammengefasst sind (siehe Abbildung 6.1).

Abbildung 6.1: Aufbau eines langen Instruktionswortes bestehend aus n einzelnen Befehlen

Die Abarbeitung der einzelnen Befehle eines langen Instruktionswortes erfolgt parallel. Dafür müssen insgesamt so viele Ausführungspfade existieren wie Befehle im langen Instruktionswort zusammengefasst sind. Beispiele für VLIW-Prozessoren sind der IA-64 von Intel und HP [HMR+00], der Itanium 2 Pro-

zessor [SR03] von Intel und der TMS320C6XX [SS99] und der TM3270 MP [vdWVD+05] von Texas Instruments. Weiterhin werden VLIW-Architekturen auch in Grafikkarten eingesetzt [JDPK09].

6.1 Aufbau der VLIW-Architektur

Bei der verwendeten Architektur handelt es sich um ein generisches Modell, welches für verschiedene Algorithmen konfiguriert werden kann. Durch eine vorherige Entwurfsraumexploration kann jeweils eine geeignete Architektur bestimmt werden, welche beispielsweise einen speziellen Signalverarbeitungsalgorithmus realisiert [Jun11, Sch06].

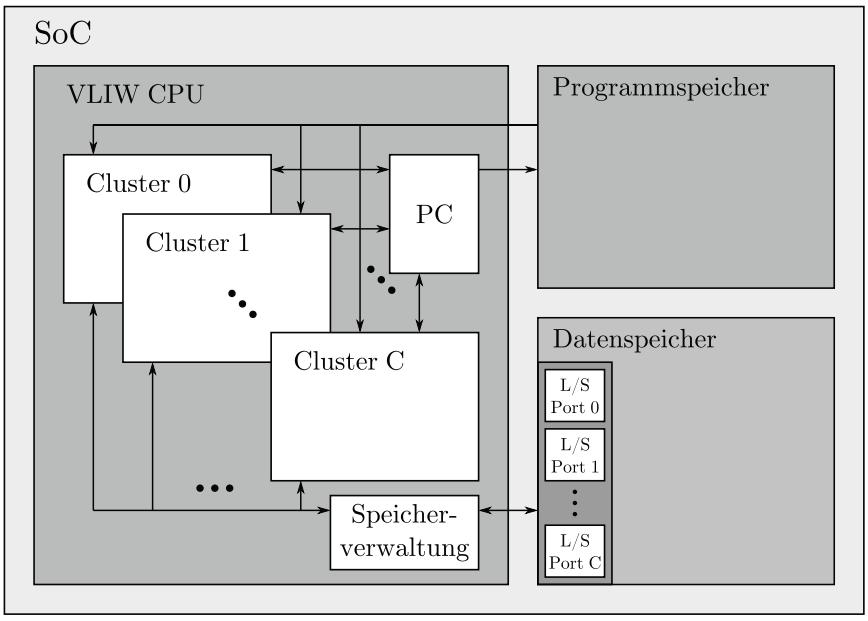


Abbildung 6.2: Schematische Darstellung des verwendeten SoC bestehend aus VLIW-Prozessor, Programm- und Datenspeicher

In Abbildung 6.2 ist der verwendete VLIW-Prozessor einschließlich des Programm- und Datenspeichers als SoC schematisch dargestellt. Das generische Modell des Prozessors besteht aus einer frei wählbaren Anzahl C identischer Cluster. Zusätzlich gibt es für den gesamten Prozessor einen zentralen Programmzähler (PC), welcher die Adressierung des Programmspeichers vornimmt. Jeder Cluster kann den Inhalt des PC Lesen und Schreiben. Hierbei muss durch den Compiler

sichergestellt sein, dass das Schreiben nur durch einen Cluster veranlasst wird. Weiterhin gibt es eine zentrale Speicherverwaltung, welche für jeden Cluster pro Takt eine Lese- oder Schreibinstruktion auf den Datenspeicher umsetzten kann. Dazu besitzt der Datenspeicher insgesamt C Lese/Schreibe (L/S) Ports. Für den Zugriff auf den Datenspeicher muss der Compiler ebenfalls sicherstellen, dass jeder Cluster nur eine Lese- beziehungsweise Schreibeinstruktion pro Instruktionswort vorsieht. Der Datenaustausch zwischen den einzelnen Clustern kann ausschließlich über Schreib- und Lesebefehle auf den Datenspeicher erfolgen.

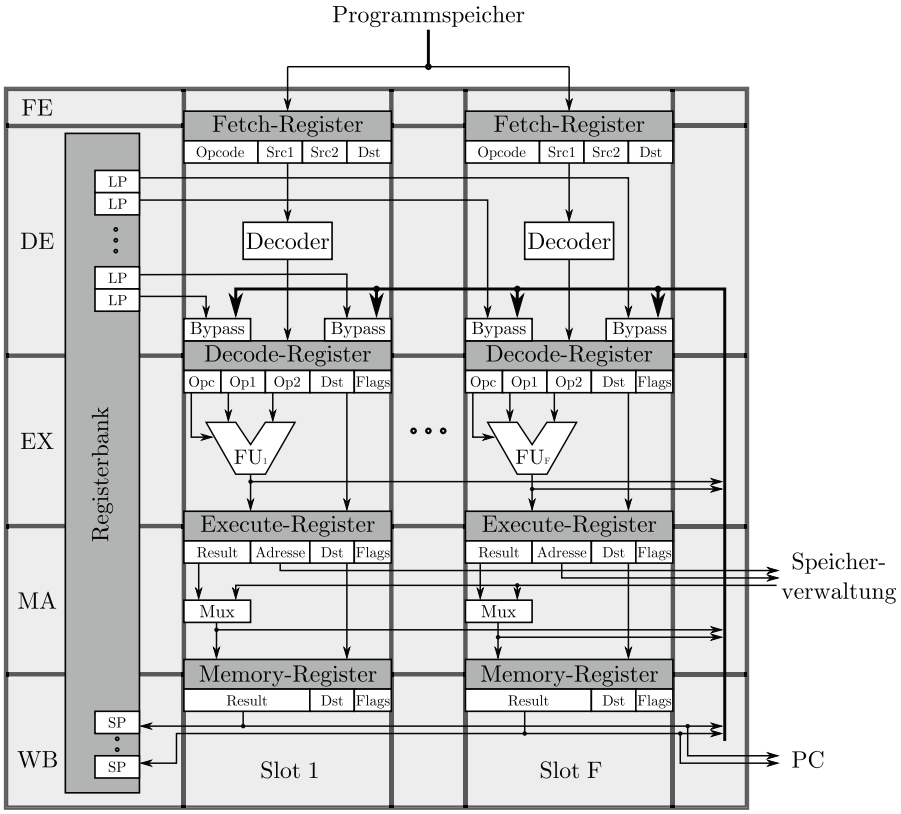


Abbildung 6.3: Schematische Darstellung eines Clusters des VLIW-Prozessors

Der schematische Aufbau eines Clusters ist in Abbildung 6.3 dargestellt. Jeder Cluster besitzt F Ausführungspfade (Slots), welche jeweils eine Funktionseinheit beinhalten und sich eine gemeinsame Registerbank teilen. Die Registerbank besteht aus 2^R Registern, wobei $R \in \mathbb{N}; R \geq 4$ ist. Die Datenbreite der einzelnen Register und aller Datenwerte beträgt 2^W Byte wobei $W \in \mathbb{N}^+$ ist. Die Datenbreite der F Funktionseinheiten ist ebenfalls 2^W Byte. Jede Funktionseinheit kann 2 Operanden verarbeiten, die durch insgesamt 2 Leseports (LP) pro Ausfüh-

rungspfad bereitgestellt werden. Insgesamt existieren also $2 \cdot C \cdot F$ Leseports die jeweils einen Operanden aus einem der 2^R Register der jeweiligen Registerbank auswählen und dem zugehörigen Ausführungspfad bereitstellen.

Die Abarbeitung der Befehle wird durch eine 5-phasige Pipeline realisiert. Die einzelnen Phasen sind dabei *fetch*, *decode*, *execute*, *memory access* und *write back* (FE,DE,EX,MA,WB). In der FE-Phase wird ein langes Instruktionswort aus dem Programmspeicher geholt und jede einzelne Instruktion wird dabei in das Fetch-Register des jeweiligen Ausführungspfads eines Clusters geladen. Die Adressierung des Programmspeichers erfolgt durch den zentralen Programmzähler (PC). Die DE-Phase dekodiert den jeweiligen Befehl aus dem Fetch-Register und stellt dabei die benötigten Operanden aus der Registerbank über die zugehörigen Leseports oder über eine Bypass-Struktur bereit. Der dekodierte Befehl wird anschließend im Decode-Register gespeichert. Dieses enthält die jeweiligen Operanden und den dekodierten Operationscode für die funktionale Einheit (FU). Weiterhin werden das Zielregister des Befehls und zusätzliche Flags für Lade-/Speicher- und Sprunginstruktionen hinterlegt. In der anschließenden EX-Phase verarbeitet die FU die jeweiligen Operanden mit dem dazugehörigen Operationscode. Das Ergebnis der Operation wird im Execute-Register gespeichert. Auch hier werden zusätzlich wieder Zielregister und die Flags aus dem Decode-Register zwischengespeichert. In der folgenden MA-Phase wird durch die zentrale Speicherverwaltung ein Lese- oder Schreibzugriff pro Cluster auf den gemeinsamen Datenspeicher realisiert. Alle notwendigen Informationen, wie Ergebnis der FU oder geladenes Datum, speichert das Memory-Register am Ende der MA-Phase. In der WB-Phase wird jede Registerbank im jeweiligen Zielregister mit dem entsprechendem Datum aktualisiert. Auch Aktualisierungen des PC, bei Sprunginstruktionen, werden in der WB-Phase umgesetzt.

Zur Steigerung der Verarbeitungsleistung des Prozessors existieren 3 Bypässe, die Datenwerte direkt aus einzelnen Pipeline-Phasen eines Clusters in der DE-Phase desselben Clusters bereitstellen. Somit stehen Ergebnisse vor dem Rückschreiben in die Registerbank zur Verfügung. Die implementierten Bypässe dieser Architektur sind folgende:

- EX-DE Bypass
- MA-DE Bypass
- WB-DE Bypass

Die einzelnen Bypässe greifen jeweils das Datum vor dem Speichern in die Pipeline-Register EX,MA,WB(bzw. Registerbank) ab und stellen dies der DE-

Phase zur Verfügung. In der DE-Phase wird die Wahl eines Datums über einen Bypass entschieden. Dies geschieht genau dann, wenn der aktuelle Befehl in der DE-Phase den Inhalt eines Quellregisters benötigt, welcher noch nicht in der Registerbank vorliegt jedoch nach vollständiger Abarbeitung des Befehls in dieses Register geschrieben wird. In den Quellcodes 6.1, 6.2 und 6.3 sind jeweils Assembleranweisungen dargestellt, die die jeweiligen Bypässe auslösen. Es handelt sich dabei um Drei-Adressen-Code mit der folgenden Kodierung "**Opcode Src1; Src2; Dst;**".

Quellcode 6.1: EX-DE Bypass

```
1 ADD R1, R2, R3;
2 ADD R3, R4, R5;
3 NOP;
4 NOP;
```

Quellcode 6.2: MA-DE Bypass

```
1 ADD R1, R2, R3;
2 NOP;
3 ADD R3, R4, R5;
4 NOP;
```

Quellcode 6.3: WB-DE Bypass

```
1 ADD R1, R2, R3;
2 NOP;
3 NOP;
4 ADD R3, R4, R5;
```

Durch die Struktur des Prozessors und seiner Befehls-Pipeline sind Daten-, und Steuerkonflikte nicht ausgeschlossen. Als Datenkonflikte sind hier nur RAW (read-after-write) Konflikte möglich. Durch die eben beschriebenen Bypässe können alle RAW-Konflikte, bis auf den folgenden, durch die Architektur selber aufgelöst werden. Quellcode 6.4 zeigt diesen Konflikt in Assembler Code.

Quellcode 6.4: RAW Konflikt

```
1 LOAD R1, R3;
2 ADD R3, R4, R5;
3 NOP;
```

Quellcode 6.5: MA-DE Bypass

```
1 LOAD R1, R3;
2 NOP;
3 ADD R3, R4, R5;
```

Der Ladebefehl in Quellcode 6.4 lädt ein neues Datum aus dem Datenspeicher in Register **R3**. Der anschließende Additionsbefehl benötigt dieses Datum als Quelloperand. Allerdings ist das Datum erst in der MA-Phase verfügbar. Dieser Konflikt muss durch den Compiler aufgelöst werden. Quellcode 6.5 löst den eben beschriebenen Konflikt mit Hilfe eines zusätzlichen NOP-Befehls und anschließender Nutzung des MA-DE Bypasses. Sämtliche durch Spungbefehle ausgelöste Steuerkonflikte in dieser Architektur müssen ebenfalls durch den Compiler aufgelöst werden.

6.1.1 Befehlssatzarchitektur

Der Befehlssatz des verwendeten VLIW-Prozessors ist an einen MIPS Befehlssatz angelehnt und beinhaltet 28 Befehle, die durch insgesamt 5 Befehlsklassen abgebildet werden. Es handelt sich hier um einen RISC Befehlssatz, bei dem alle Befehle eine identische Länge haben. Somit besitzt jedes lange Instruktionswort des Prozessors genau $C \cdot F$ Befehle identischer Länge. Die Befehlsklassen unterscheiden sich in Funktion und Kodierung und sind im folgenden aufgelistet.

NOP-Befehl

Diese Befehlsklasse beinhaltet nur den NOP-Befehl. Dieser stellt eine Leeroperation dar, welche rein funktional keine Bedeutung hat und demzufolge auch keine Veränderung der Register aus der Registerbank veranlasst. Ein NOP-Befehl wird in der DE-Phase durch den Befehlscode 00000000 erkannt. Während der Ausführung wird er einfach, ohne funktionale Bedeutung, durch die Pipeline geschoben.

Register - Register - Befehle

Befehle dieser Klasse verwenden Registerinhalte aus der zentralen Registerbank des jeweiligen Clusters oder durch die Bypass-Struktur zur Berechnung von arithmetischen und logischen Operationen. Die Ergebnisse dieser Operationen werden wieder in der zentralen Registerbank gespeichert. Befehle dieser Befehlsklasse adressieren je nach Funktion ein oder zwei Quellregister und ein Zielregister.

Immediate - Befehle

Diese Klasse beinhaltet für die verwendete Architektur ebenfalls nur einen Befehl mit dem ein Wert, kodiert im Befehl, in die Registerbank geschrieben werden kann.

Sprung - Befehle

Diese Befehlsklasse beinhaltet sämtliche Instruktionen, die einen Sprung im Programmcode realisieren können. Dazu zählen zum einen bedingte und unbe-

dingte Sprünge sowie Funktionsaufrufe, welche zusätzlich den Inhalt des PC in einem beliebigen Register der Registerbank sichern.

Lade- / Speicher - Befehle

Die letzte Klasse von Befehlen dieser Architektur beinhaltet alle Instruktionen, welche einen Zugriff auf den Datenspeicher realisieren. Alle Befehle dieser Klasse sind in dieser Architektur die einzige Möglichkeit des Datenaustausches zwischen den Clustern.

Sämtliche Befehle dieser Architektur haben eine einheitliche Länge von 8-Bit Befehlscode und $3 \cdot R$-Bit für mögliche Registeradressierungen. Diese Strukturierung führt allerdings dazu, dass einige Befehle einen sehr hohen Anteil an *don't care*-Bits besitzen. Andere VLIW-Prozessoren verwenden eine variable Befehlscodierung, um den Nachteil des hohen Programmspeicherbedarfs zu reduzieren [HMR+00]. Diese zieht dann wieder einen höheren Aufwand für die Dekodierung nach sich. Die jetzige Kodierung der Befehle erlaubt es, für diese Architektur zahlreiche Erweiterungen des Befehlssatzes zu realisieren. Beispielsweise ist es problemlos möglich, die Klasse der Immediate-Befehle um zusätzliche arithmetische Operationen mit Konstanten zu erweitern. Dafür muss lediglich der Dekoder der DE-Phase modifiziert werden. Ebenso sind Erweiterungen im Bereich der Lade- und Sprung-Befehle denkbar, welche entweder zusätzliche Adressierungsmodi oder Sprungbedingungen nutzen. In vielen Fällen ist auch hier eine Modifikation der DE-Phase ausreichend. Die in dieser Arbeit implementierte Architektur ist angelehnt an den VLIW-Prozessor aus [Sch06]. Der vollständige Befehlssatz der verwendeten Architektur befindet sich im Anhang dieser Arbeit in Tabelle A.1.

6.2 Erweiterungen der Architektur

Zur Umsetzung des Verfahrens dieser Arbeit ist der vorgestellte VLIW-Prozessor zu erweitern. Eine, nach der vorgestellten Herangehensweise aus Kapitel 4 Abbildung 4.16, Untersuchung der einzelnen Komponenten auf Register-Transfer-Ebene hat ergeben, dass lediglich Redundanz der funktionalen Einheiten durch die vorgestellte Implementierung der RLB sinnvoll ist. Aus diesem Grund wurde die Architektur auf RT-Ebene so verändert, dass alle funktionalen Einheiten zu einem RLB zusammengefasst wurden. Aufgrund der aufgezeigten hohen Temperaturabhängigkeit von Alterungsfehlern in Abschnitt 2.3.2 und der Ausbildung von thermischen *hotspots* in funktionalen Einheiten von VLIW-Prozessoren [MLN+06] ist der Einsatz von Redundanz für diese Komponenten besonders relevant. Zusätzlich zur Redundanz für funktionale Einheiten wurde der zur Überwachung von Test, Diagnose und Rekonfiguration benötigte RLB-Controller auf RT-Ebene ergänzt.

In Abbildung 6.4 sind die Erweiterungen des VLIW-Prozessors für eine Konfiguration mit 2 Clustern und jeweils 2 Ausführungspfaden schematisch dargestellt. Zusätzlich zum RLB-Controller und dem zusätzlichen Lese-/Schreibe-Port des Datenspeichers ist ebenfalls der auf RT-Ebene implementierte RLB und der dazugehörige RLB-Speicher für den Status der RLBs abgebildet. Der RLB erstreckt sich über alle ursprünglichen funktionalen Einheiten aller Cluster und beinhaltet ebenfalls redundante FUs zur Realisierung der vorgestellten Ersetzungsstrategie.

6.2.1 Implementierung der Redundanz

Der vorgestellte generische VLIW-Prozessor wurde bezüglich der Redundanz so erweitert, dass sämtlichen funktionalen Einheiten zu einem RLB zusammengefasst wurden. Die Implementierung des RLB wurde so umgesetzt, dass jeder FU eine beliebige Anzahl an zusätzlichen Ersatz-FU zugeteilt werden kann. Auch die vorgestellte Segmentierung wurde in der Implementierung realisiert. Mögliche Segmentierungen sind abhängig von der Gesamtzahl vorhandener funktionaler Einheiten der gesamten Architektur. Die Umsetzung einer bestimmten RLB-Konfiguration kann durch zwei Konstanten in der VHDL Bibliothek definiert werden.

- $RLB\_number\_of\_spares\_per\_segment := X$ - definiert die Anzahl der Ersatz-FU die in jedem Segment zur Verfügung stehen

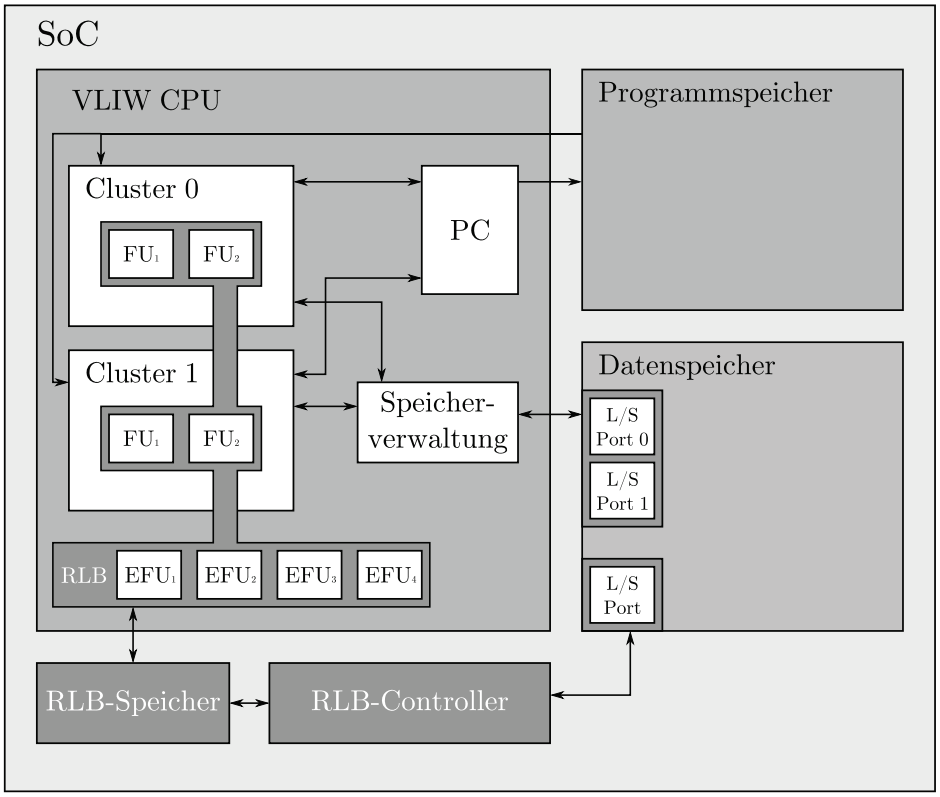


Abbildung 6.4: Schematische Darstellung des erweiterten SoC für eine Konfiguration des VLIW-Prozessors bestehend aus 2 Clustern mit jeweils 2 Ausführungspfaden und 4 Ersatz-FUs

- $RLB\_original\_fus\_per\_segment := Y$ - definiert die Menge der ursprünglichen FU, welche zu einem Segment zusammengefasst werden
- dabei muss gelten $M/Y \in \mathbb{N}^+$
- die Anzahl der Segmente P einer RLB-Konfiguration ergibt sich dann stets aus $P = M/Y$

In Tabelle 6.1 sind alle möglichen Segmentierungen für drei Beispielarchitekturen mit $M = 9, 12$ bzw. 15 ursprünglichen funktionalen Einheiten im gegebenen System aufgelistet. Dabei ist die Interpretation der Schreibweise für eine RLB-Konfiguration folgende: $[P] \cdot ([Y] + [X])$

Diese Implementierung bietet eine umfangreiche Möglichkeit verschiedene RLB-Konfigurationen für die funktionalen Einheiten dieses VLIW-Prozessors, unab-

mögliche RLB-Konfigurationen		
(für $M = 9$)	(für $M = 12$)	(für $M = 15$)
$1 \cdot (9 + X)$	$1 \cdot (12 + X)$	$1 \cdot (15 + X)$
–	$2 \cdot (6 + X)$	–
$3 \cdot (3 + X)$	$3 \cdot (4 + X)$	$3 \cdot (5 + X)$
–	$4 \cdot (3 + X)$	–
–	–	$5 \cdot (3 + X)$
–	$6 \cdot (2 + X)$	–
$9 \cdot (1 + X)$	–	–
–	$12 \cdot (1 + X)$	–
–	–	$15 \cdot (1 + X)$

Tabelle 6.1: Beispielkonfigurationen des RLBs für verschiedene Konfigurationen des VLIW-Prozessors mit $M = 9, 12, 15$ ursprünglichen funktionalen Einheiten

hängig von den gewählten generischen Parametern, umzusetzen. Die Anzahl zusätzlicher Ersatz-FU ist abhängig von der gewählten Segmentierung und durch die Anzahl der gegebenen funktionalen Einheiten einer Konfiguration des VLIW-Prozessors eingeschränkt.

6.2.2 Administration von Test, Diagnose und Redundanz

Zentrales Element zur Realisierung der Verwaltung redundanter Baugruppen und des Tests, einschließlich Diagnose, dieser Arbeit ist der RLB-Controller. Diese Baugruppe steuert den Test für die einzelnen funktionalen Einheiten des RLB und realisiert dessen Rekonfiguration im Falle eines Fehlers. Zusätzlich muss der RLB-Controller eine erschöpfte Redundanz nach außen Kommunizieren. Der RLB-Controller wurde in VHDL generisch beschrieben und passt sich damit der gewählten RLB-Konfiguration an. Die Aufgaben und die Funktionsweise dieser Komponente wurden bereits in Abschnitt 4.1.5 beschrieben. Zusätzlich zu der beschriebenen Funktionsweise besitzt der RLB-Controller Zugriff auf den Datenspeicher des VLIW-Prozessors. Durch diese Zugriffsfunktion auf den Datenspeicher wird für dieses Anwendungsbeispiel die Interaktion zwischen dem softwarebasierten Selbsttest und dem RLB-Controller realisiert. Damit benötigt der Datenspeicher einen weiteren Lese-/Schreib- Port.

Interaktion von RLB-Controller und SBST

Um den im Abschnitt 4.2.2 vorgestellten Systemablauf mit einem SBST umzusetzen, benötigt der RLB-Controller die Möglichkeit die Abarbeitung einer dedizierten Test-Software zu Starten und deren Ergebnisse abzufragen und auszuwerten, um gegebenenfalls einen neuen Status des RLB über das RLB-Status Register zu veranlassen. Nach einem erfolgreichen Test muss der RLB-Controller zusätzlich in der Lage sein, den VLIW-Prozessor in den Normalbetrieb zu versetzen und die Abarbeitung der Nutzer-Software zu veranlassen.

Quellcode 6.6: Programmspeicher

```
// Inhalt Programmspeicher
// Initialisierung
LOAD R0, #START ; ...
NOP             ; ...
JMP R0          ; ...
...
// SBST für C=0 F=0
SBST_C-0_F-0:
...
...
// SBST für C=C F=F
SBST_C-C_F-F:
...
// Nutzer Software
USER_SOFTWARE:
...
...
```

Quellcode 6.7: Datenspeicher

```
// Inhalt Datenspeicher
// Steuerinformationen
// für den RLB-Controller
// Einsprungziele:
START: #USER_SOFTWARE;
#SBST_C-0_F-0;
...
#SBST_C-C_F-F;
#USER_SOFTWARE;
// Dauer SBST
SBST_TIME: <VALUE>
// Test Intervall
SW_TEST_INTERVALL: <VALUE>
// Erwartete Testantwort
SBST_GOOD: <GOOD_SIGNATURE>
// Testantwort des SBST
SBST_SIM: <SIM_SIGNATURE>
```

In Quellcode 6.6 und 6.7 sind Inhalt von Programm- und Datenspeicher für dieses Anwendungsbeispiel aufgelistet, mit dem der RLB-Controller in die Lage versetzt wird, verschiedene Programme auszuführen. Gleichzeitig beinhaltet der Datenspeicher zusätzliche Informationen, wie die erwartete Testantwort des SBST und die notwendige Ausführungszeit.

Ablauf bei Systemstart

Nach einem Systemstart liest der RLB-Controller aus dem Datenspeicher die Information über die Dauer eines SBST (`SBST_TIME <VALUE>`) aus und initialisiert einen internen Zähler. Anschließend wird das Label `START` auf den nächsten auszuführenden SBST geändert. Danach wird die Abarbeitung des VLIW-

Prozessors freigegeben und dieser führt die gewünschte Test-Software aus. Nach dem vollständigen Durchlauf des ersten SBST wird die erstellte Signatur im Datenspeicher an die Adresse `SBST_SIM` geschrieben. Darauf folgend unterbricht der RLB-Controller die Abarbeitung und kann durch einen Vergleich der Speicherinhalte des Datenspeichers der Adressen `SBST_GOOD` und `SBST_SIM` das Testergebnis bestimmen. Bei identischen Werten wird der Inhalt von Adresse `SBST_SIM` zurückgesetzt und das Label `START` auf die nächste Test-Software oder, nach vollständigem Test, auf die Nutzer-Software geändert. Bei unterschiedlichen Werten wird ein neuer Status des RLB veranlasst und der Test wiederholt. Führt eine wiederholte Rekonfiguration des RLB zu einer erschöpften Redundanz hält der RLB-Controller den VLIW-Prozessor an und signalisiert den fehlerhaften Zustand nach außen.

Ablauf im Anwendungsfall

Um eine Kompensation von permanenten Fehlern im laufenden Betrieb zu realisieren, wurden in Abschnitt 4.3 zwei Varianten beschrieben. Dazu zählte zum ersten ein geplanter Test in Leerlaufphasen, der durch die Nutzer-Software selbst veranlasst wird, und zum zweiten ein ungeplanter Test, wenn die Nutzer-Software durch möglich Fehler einen geplanten Test nicht mehr veranlassen kann. Beide Szenarien wurden für dieses Anwendungsbeispiel umgesetzt und lassen sich wie folgt zusammenfassen.

Nach einer erfolgreichen Absolvierung der Test-Software nach dem Systemstart wird der interne Zähler des RLB-Controllers mit dem Wert des Datenspeichers an der Adresse `SW_TEST_INTERVALL` initialisiert. Nach Ablauf des Zählers im RLB-Controller hält dieser den VLIW-Prozessor an und liest das Datum an Adresse `SBST_SIM` aus. Dieses Datum beinhaltet eine von drei möglichen Informationen.

- `IDLE` - die Software befindet sich in einer Leerlaufphase und der Prozessorstatus wurde gesichert
- `ALIVE` - die Software befindet sich nicht in einer Leerlaufphase, arbeitet aber wie geplant und hat dies mit dem zyklischen Schreiben des Wertes `ALIVE` bestätigt
- nicht `IDLE/ALIVE` - bei der Ausführung der Software ist ein Fehler aufgetreten, welcher ein ungewünschtes Verhalten zur Folge hatte (die Software konnte den Wert `ALIVE/IDLE` nicht im Datenspeicher aktualisieren)

Erkennt der RLB-Controller den Wert `IDLE`, so wird ein Test wie beim Systemstart durchgeführt. Im fehlerfreien Fall wird die Nutzer-Software, durch Wiederherstellung des Prozessorstatus, fortgesetzt. Im Falle eines Fehlers und damit einer Rekonfiguration des Systems muss die Nutzer-Software neu gestartet oder von einem validen Checkpunkt fortgesetzt werden, da eine Kompromittierung des Prozessorstatus nicht ausgeschlossen ist. Die Erstellung eines solchen Checkpunktes ist nicht Bestandteil dieser Arbeit. Findet sich der Wert `ALIVE` an der erwarteten Adresse im Datenspeicher, wird dieser gelöscht und der interne Zähler des RLB-Controller erneut initialisiert. Die Abarbeitung der Nutzer-Software wird regulär fortgesetzt. Wird keiner der beiden Werte durch den RLB-Controller ausgelesen, ist von einem Fehler im laufenden Betrieb auszugehen und der Test mit anschließender Rekonfiguration wird folglich veranlasst. Auch in diesem Fall ist die Software neu zu starten beziehungsweise ab einem validen Checkpunkt zu wiederholen.

6.3 Entwurf und Validierung des SBST

Die Umsetzung von Test und Diagnose für das Anwendungsbeispiel in dieser Arbeit wurde durch einen softwarebasierten Selbsttest realisiert. Dieser Abschnitt beschreibt die Erstellung und Validierung der Test-Software. Die Erstellung des SBST wurde durch ein ATPG gestütztes Verfahren durchgeführt. Dafür sind entsprechende Testmuster für die funktionalen Einheiten generiert und anschließend auf Software-Templates abgebildet worden. Abbildung 6.5 zeigt den Entwurfsprozess für den SBST.

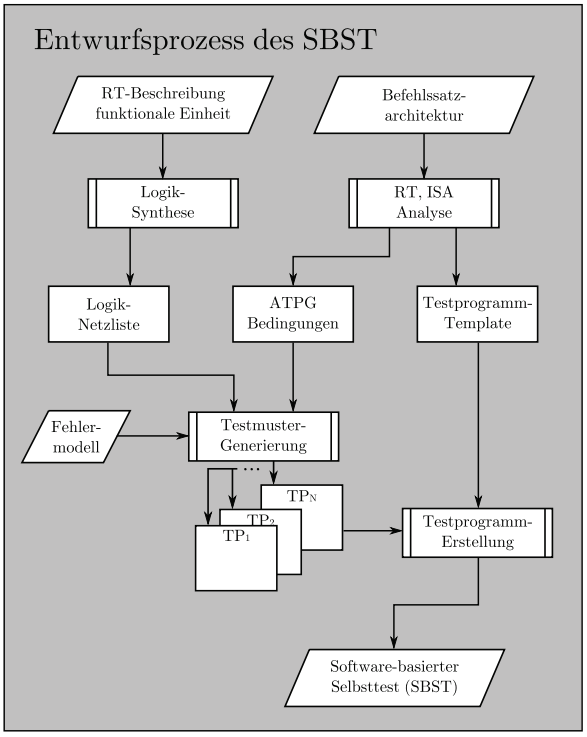


Abbildung 6.5: Entwurfsprozess des softwarebasierten Selbsttest

Nach der Synthese der funktionalen Einheit wird die erzeugte Netzliste einem ATPG-Programm bereitgestellt. Anschließend werden Testmuster für das Transitions- und Haftfehlermodell erzeugt, die eine bestimmte strukturelle Fehlerüberdeckung der FU erreichen. Jedes Testmuster enthält Werte für die beiden Operanden und den jeweiligen Opcode der funktionalen Einheit. Diese Testmuster sind nun auf geeignete Sequenzen von Assemblerbefehlen abzubilden. In den Quellcodes 6.8 und 6.9 sind jeweils Assemblerroutinen für das Haft- und Tran-

sitionsfehlermodell für den VLIW-Prozessor in einer Konfiguration mit einem Cluster mit zwei Ausführungspfaden dargestellt, welche jeweils ein Testmuster an die funktionale Einheit des ersten Ausführungspfades applizieren. Die Werte beider Operanden befinden sich in diesem Beispiel jeweils im Datenspeicher und der Opcode wurde direkt als Assemblerbefehl in das Template eingefügt.

Quellcode 6.8: Assemblerroutine für das Haftfehlermodell

```
// Inhalt Programmspeicher
// Initialisierung
LDC 1,      R0; NOP
// Lade 1. Testmuster
// Lade 1. Operanden
LOAD R0,    R1; NOP
// Erhöhe Adresse
INC         R0; NOP
// Lade 1. Testmuster
// Lade 2. Operanden
LOAD R0,    R2; NOP
// Erhöhe Adresse
INC         R0; NOP
// Initialisiere 1.
// Testmuster
ADD R1, R2, R3; NOP
// Speichere Testantwort
STORE R0,   R3; NOP

// Inhalt Datenspeicher
// 1. Testmuster
// 1. Operand
0110101010101010
// 2. Operand
0111010100111110
// ermittelte Testantwort
0000000000000000
// erwartete Testantwort
0100100100110000
```

Quellcode 6.9: Assemblerroutine für das Transitionsfehlermodell

```
// Inhalt Programmspeicher
// Initialisierung
LDC 1,      R0; NOP
// Lade 1. Testmuster
// Lade 1. Operand
LOAD R0,    R1; NOP
// Erhöhe Adresse
INC         R0; NOP
// Lade 2. Operanden
LOAD R0,    R2; NOP
// Erhöhe Adresse
INC         R0; NOP
// Lade 2. Testmuster
// Lade 1. Operanden
LOAD R0,    R3; NOP
// Erhöhe Adresse
INC         R0; NOP
// Lade 2. Operanden
LOAD R0,    R4; NOP
// Erhöhe Adresse
INC         R0; NOP
// Init. 1. Testmuster
ADD R1, R2, R5; NOP
// Init. 2. Testmuster
ADD R3, R4, R6; NOP
// Speichere Testantwort
STORE R0,   R6; NOP

// Inhalt Datenspeicher
// 1. Testmuster 1. Operand
0110101010101010
// 2. Operand
0111010100111110
// 2. Testmuster 1. Operand
1110101010101000
// 2. Operand
0101111010001000
// ermittelte Testantwort
0000000000000000
// erwartete Testantwort
0100100100110000
```

Die Umsetzung der Testmuster mit den Assemblerroutinen in Quellcodes 6.8 und 6.9 zeigt die grundsätzliche Realisierung des SBST. Allerdings würde diese Umsetzung für alle Testmuster einen sehr hohen Speicherbedarf bedeuten, da jedes Testmuster für das Haftfehlermodell 7 Zeilen Programm- und 4 Zeilen Datenspeicher und für das Transitionsfehlermodell 12 Zeilen Programm- und 6 Zeilen Datenspeicher belegt. Zusätzlich müsste der RLB-Controller die Testantworten jedes applizierten Testmusters mit den Testantworten des ATPG-Programms vergleichen.

Um sowohl den Speicherbedarf des Software-Tests als auch die Laufzeit des Vergleichs durch den RLB-Controller zu reduzieren, wurde der SBST modifiziert. Die modifizierte Variante des SBST für das Transitionsfehlermodell ist als Pseudocode in Quellcode 6.10 zu sehen.

Quellcode 6.10: Speicheroptimierte Variante des SBST zur Applizierung von Testmustern für Transitionsfehler

```
1  # Inhalt Programmspeicher
2  # Initialisierung
3  LDC #TEST_DATA_BEGIN,   R0;
4
5  # Durchlaufe alle Testmuster
6  for i in 0 to #TESTMUSTER do
7          # Lade 1. Operanden vom 1. Testmuster in R1
8          LOAD R0,        R1;
9          INC             R0;
10         # Lade 2. Operanden vom 1. Testmuster in R2
11         LOAD R0,        R2;
12         INC             R0;
13         # Lade 1. Operanden vom 2. Testmuster in R3
14         LOAD R0,        R3;
15         INC             R0;
16         # Lade 2. Operanden vom 2. Testmuster in R4
17         LOAD R0,        R4;
18         INC             R0;
19         # Lade Opcode in Register R5
20         LOAD R0,        R5;
21         # Berechne Sprungziel für die Test Ausführung
22         R6 = sprungziel(R5);
23         # Springe zur Testausführung an Adresse in R6
24         CALL R6, R7;
25         # Testantwort ist in Register R8
26         # Bilde die Signatur in Register R9
27         ADD R9, R9, R8;
28 od
29
30 # Speichere die gebildete Signatur
31 STORE #SBST_SIM,        R9;
```

Diese Implementierung durchläuft eine Schleife für alle Testmuster, welche als erstes alle Operanden und den jeweiligen Opcode in interne Register lädt. Anschließend wird das Sprungziel berechnet, an dem der Test durch Applizieren des Testmusters ausgeführt wird (Zeile 22 in Quellcode 6.10). Für jeden Opcode existiert ein Sprungziel, an dem der jeweilige Befehl, der diesen Opcode mit den eben geladenen Registern verwendet und damit das Testmuster an die FU anlegt, gespeichert ist. Durch diese Umsetzung des SBST wird eine feste Anzahl an Zeilen im Programmspeicher benötigt, die unabhängig von der Anzahl der zu applizierenden Testmuster ist. Die Akkumulation der Testantworten jedes Testmusters durch den SBST erzeugt eine Signatur am Ende des jeweiligen Tests, welche dann schnell durch den RLB-Controller mit dem erwarteten Ergebnis im Datenspeicher `SBST_GOOD` verglichen werden kann. Der benötigte Platz für die einzelnen Testmuster im Datenspeicher ergibt sich weiterhin aus der Anzahl der benötigten Testmuster.

Die Akkumulation der Testantworten zu einer einzelnen Signatur für den gesamten SBST verringert den Aufwand für den Vergleich durch den RLB-Controller. Jedoch ist durch diese Umsetzung nicht ausgeschlossen, dass fehlerhafte Testantworten durch die Bildung einer Signatur verloren gehen. Somit kann die gegebene Fehlerüberdeckung der Testmuster durch das ATPG-Programm von der realen Fehlerüberdeckung des SBST abweichen. Zur Ermittlung der tatsächlichen Fehlerüberdeckung des SBST muss das erstellte Testprogramm durch eine Simulation mit einer Fehlerinjektion validiert werden. In Abbildung 6.6 ist der Ablauf zur Validierung der tatsächlichen Fehlerüberdeckung des SBST dargestellt.

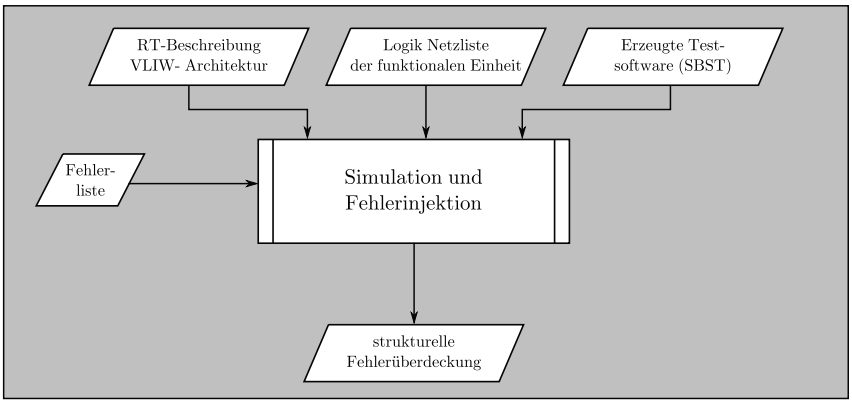


Abbildung 6.6: Validierung des SBST durch Simulation und Fehlerinjektion

Jeder Fehlerpunkt aus der Fehlerliste der jeweiligen funktionalen Einheit wird einzeln in das VHDL Modell injiziert. Anschließend wird eine vollständige Simu-

lation des jeweiligen SBST durchgeführt. Unterscheidet sich die generierte Signatur des SBST anschließend von der Referenzsignatur, so gilt der Fehlerpunkt als erkannt. Durch dieses simulationsbasierte Verfahren wird die tatsächliche Fehlerüberdeckung des Software-Tests ermittelt.

Der vollständige SBST für eine beliebige Konfiguration des VLIW-Prozessors setzt sich aus insgesamt $C \cdot F$ einzelnen Testprogrammen zusammen. Jedes Testprogramm führt alle Instruktionen für die Initialisierung und Ausführung der Testmuster sowie die Akkumulation der Testantworten zu einer Signatur im jeweiligen Ausführungspfad der zu testenden funktionalen Einheit aus. Der gewählte Aufbau des SBST kann für beliebige Konfigurationen des VLIW-Prozessors ohne Anpassungen umgesetzt werden. Jedoch steigt der benötigte Programmspeicher für das Testprogramm mit der Anzahl an Clustern und Ausführungspfaden. Für Konfigurationen des VLIW-Prozessors mit vielen Ausführungspfaden ist daher eine Zusammenlegung einzelner Testprogramme und die damit verbundene parallele Ausführung eine Möglichkeit zur Reduzierung des Bedarfes an Programmspeicher. Die Parallelisierung der Test Programme ist dann zum ersten abhängig von den zur Verfügung stehenden Registern und zum zweiten muss ebenfalls der RLB-Controller diesbezüglich angepasst werden.

6.4 Ergebnisse einer Konfiguration des VLIW-Prozessors

Dieser Abschnitt beinhaltet die Auswertung des vorgestellten Verfahrens dieser Arbeit für eine Konfiguration des in diesem Kapitel beschriebenen VLIW-Prozessors. Im ersten Teil wird auf die Flächenaufteilung der einzelnen Komponenten einer Konfiguration des VLIW-Prozessors, die ermittelte Fehlerüberdeckung des erstellten SBST sowie den notwendigen Speicherbedarf des SBST eingegangen. Anschließend folgt eine Auswertung der Auswahl von Redundanz zur Maximierung von Zuverlässigkeit, Lebensdauer und Produktionsausbeute einschließlich einer detailierten Betrachtung der sich durch eine RLB-Konfiguration ergebenden Auswirkungen auf den gesamten Prozessor. Als Anwendungsbeispiel wurde folgende Konfiguration des VLIW-Prozessors gewählt:

- Anzahl der Cluster $C = 2$
- Anzahl der Ausführungspfade pro Cluster $F = 4$
- 64-Bit Datenbreite der Ausführungspfade $W = 8$
- 32 frei verfügbare Register pro Cluster $R = 5$

Tabelle 6.2 zeigt sowohl die Anzahl als auch die beanspruchte Fläche der einzelnen Komponenten des VLIW-Prozessors nach einer RT- und Logiksynthese mit Hilfe des Encounter RTL Compiler[1] und einer freien 45nm Synthesebibliothek[2]. Diese Auflistung zeigt lediglich die Aufteilung des VLIW-Prozessors in seine einzelnen Komponenten ohne Programm- und Datenspeicher. Auch die verwendete Registerbank wurde mit Hilfe des Synthesewerkzeuges erstellt und besteht nicht aus einem SRAM Speicherbaustein.

Durch die gegebene Konfiguration des VLIW-Prozessors konnte, wie im vorherigen Abschnitt 6.4 beschrieben, ein softwarebasierter Selbsttest für die funktionalen Einheiten entwickelt werden. In Tabelle 6.3 sind die Fehlerpunkte einer FU für das Haft- sowie das Transitionsfehlermodell und die daraus resultierende Fehlerüberdeckung aufgelistet. Es wurden lediglich Testmuster für das Transitionsfehlermodell erzeugt. Die resultierende Fehlerüberdeckung für das Haftfehlermo-

[1] Synthese- und Layout-Werkzeug der Firma Cadence Design Systems, Inc (Version v07.20-s009_1 (64-Bit))

[2] NANGATE Open Cell Library - PDKv1.3_2010_12 (released August 2011)

Flächenverteilung der gewählten Konfiguration des VLIW-Prozessors			
Komponente	**Anzahl**	**$\sum$ Fläche in μm^2**	**Fläche in %**
VLIW-Prozessor	1	386 800	100.000
Programmzähler (PC)	1	1 164	0.301
Speichverwaltung	1	4 097	1.059
Registerbank	2	40 704	10.523
FE (Fetch)	8	1 376	0.356
DE (Decode)	8	56 191	14.527
EX (Execute)	8	6 096	1.576
FU	8	271 132	70.096
MA (Memory Access)	8	6 040	1.562

Tabelle 6.2: Flächenverteilung für die gewählte Konfiguration

Fehlerüberdeckung des SBST		
	Fehlermodell	
	Haftfehler	**Transitionsfehler**
Testmuster	1483	
alle Fehlerpunkte	109 834	153 790
erkannte Fehlerpunkte	109 441	151 039
Fehlerüberdeckung (einzeln)	99.642 %	98.211 %
Fehlerüberdeckung (insgesamt)	**98.807 %**	

Tabelle 6.3: Zusammensetzung der Fehlerüberdeckung für eine funktionale Einheit mit 64-Bit Verarbeitungsbreite

dell wurde mit den Testmustern des Transitionsfehlermodells erreicht. Die Testmuster wurden durch den Testmustergenerator Tessent[3] erzeugt. Der notwendige Parameter für die gezielte Auswahl von Redundanz beträgt nach beschriebener Validierung des SBST $FC = 98.807\%$.

Der resultierende Speicheraufwand für den softwarebasierten Selbsttest der gewählten Konfiguration ist in Tabelle 6.4 zusammengefasst. Dieser ist unterteilt in den Aufwand im Programm- und Datenspeicher. Dabei beschreibt die erste Zeile der Tabelle die benötigten *lines of code* (LOC) für den SBST. Hier entspricht die Anzahl der Zeilen der Anzahl der verwendeten langen Instruktionswörter des gesamten SBST. Anschließend findet sich der gesamte Speicheraufwand gemessen in Kilobyte (kB).

Durch Logiksynthese mit anschließendem Layout des ursprünglichen Systems und Ermittlung der Fehlerüberdeckung wird für die Entwurfsraumexploration

[3]Testmustergenerator der Firma Mentor Graphics (Version v9.4)

Speicherbedarf des SBST		
	Programmspeicher	**Datenspeicher**
LOC	872	7 415
Datenmenge (kB)	19.59	57.94
Datenmenge insgesamt (kB)	**77.52**	

Tabelle 6.4: Speicherbedarf des SBST für Programm- und Datenspeicher

lediglich noch die administrative Fläche der jeweiligen RLB-Konfiguration benötigt. Diese setzt sich für dieses Anwendungsbeispiel aus dem Schaltnetzwerk, dem RLB-Controller und dem RLB-Speicher zusammen. Die jeweiligen Flächen wurden für dieses Anwendungsbeispiel durch Logiksynthese und Layout für jede RLB-Konfiguration ermittelt. Dieser zusätzliche Aufwand für Synthese und Layout kann durch Approximieren der Flächen für verschiedene RLB-Konfigurationen nach dem Verfahren in [5] stark reduziert werden.

6.4.1 Maximierung der Zuverlässigkeit

Durch Erstellung des softwarebasierten Selbsttests und Ermittlung der möglichen Fehlerüberdeckung sowie der administrativen Fläche kann jetzt eine zielgerichtete Redundanzauswahl zur Maximierung der Zuverlässigkeit erfolgen. Die Ergebnisse für diese Entwurfsraumexploration finden sich in Tabelle 6.5. Dabei beinhaltet die erste Spalte verschiedene RLB-Konfigurationen. In Spalte 2 ist der resultierende Hardware-Mehraufwand dargestellt. Die Spalten 3 bis 5 beinhalten den *reliability improvement factor* (RIF) der jeweiligen RLB-Konfigurationen bei einer zu erreichenden Zuverlässigkeit des Gesamtsystems von $R_{VLIW_{RLB}} = 0.9|0.99|0.999$. Die jeweils optimale RLB-Konfiguration ist in der jeweiligen Spalte **fett** dargestellt. Als obere Grenze für eine zusätzliche Fläche von Hardware wurde für dieses Beispiel 500% festgesetzt. Die Entwurfsraumexploration für die RLB-Konfigurationen wurde beim Übersteigen dieser Grenze für jede Segmentierung abgebrochen. Nicht dargestellte RLB-Konfigurationen innerhalb der oberen Grenze resultieren für dieses Beispiel in einem geringeren RIF.

Im folgenden werden die drei optimalen RLB-Konfigurationen der Spalten 3-5 aus Tabelle 6.5 in Bezug auf ihre Auswirkungen betrachtet. In Tabelle 6.6 ist der jeweilige Hardware-Mehraufwand im Verhältnis zum ursprünglichen System ohne Redundanz detailiert aufgelistet. Ebenfalls in Tabelle 6.6 findet sich der Einfluss der gewählten RLB-Konfigurationen auf die zusätzliche Verzögerung des Pro-

Zuverlässigkeitssteigerung der gewählten VLIW-Konfiguration				
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	RIF **für** $R_{VLIW_{RLB}} =$ 0.9	RIF **für** $R_{VLIW_{RLB}} =$ 0.99	RIF **für** $R_{VLIW_{RLB}} =$ 0.999
1x(8+1)	11.125	2.34	2.91	**3.00**
1x(8+2)	20.585	2.62	2.91	2.94
1x(8+3)	30.044	2.61	2.84	2.87
2x(4+1)	19.924	2.46	**2.93**	2.99
2x(4+2)	38.182	**2.63**	2.89	2.91
2x(4+3)	56.440	2.58	2.81	2.83
4x(2+1)	37.521	2.53	2.92	2.97
4x(2+2)	73.377	2.60	2.84	2.86
4x(2+3)	109.232	2.53	2.74	2.77
8x(1+1)	72.716	2.54	2.88	2.92
8x(1+2)	143.766	2.53	2.75	2.77

[Zahl] - RLB-Konfiguration mit maximaler Zuverlässigkeitssteigerung

Tabelle 6.5: Einfluss verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche zu erreichende Zuverlässigkeiten 0.9|0.99|0.999

zessors, da die zusätzlichen Schaltnetze für die Realisierung der Rekonfiguration eben auch Auswirkungen auf den maximalen Systemtakt haben. Zur Ermittlung der Verzögerungen der unterschiedlichen RLB-Konfigurationen wurden sowohl die Bibliotheken als auch die Einstellungen des Synthese- und Layoutwerkzeuges verwendet, welche bereits zur Ermittlung der Ergebnisse aus Tabelle 6.2 genutzt wurden.

Im letzten Abschnitt von Tabelle 6.6 ist der Einfluss der RLB-Konfigurationen auf die verschiedenen Ablaufzeiten des softwarebasierten Selbsttests dargestellt. Hier befinden sich in den ersten vier Zeilen die Dauer der Ausführung des SBST als Anzahl von Taktzyklen. Die Zeilen 1 und 2 beinhalten die Anzahl an Takten für den Test zum Systemstart und die Zeilen 3 und 4 für den Test im laufenden Betrieb. Die letzte Zeile beinhaltet die maximale Dauer der Ausführung (entsprechend der Takte in Zeile 4) des gesamten SBST bei maximalem Systemtakt in Millisekunden.

Für die gewählten RLB-Konfigurationen ist eine maximale Dauer der SBST Abarbeitung kleiner als 1 Millisekunde. Die geringen Unterschiede der Ablaufzeiten bei Systemstart und im Betrieb resultieren aus der geringen zusätzlichen Zeit zur Sicherung t_{save} und Wiederherstellung $t_{restore}$ des Prozessorstatus auf Assemblerebene.

Hardware-Mehraufwand in %			
	2x(4+2)	**2x(4+1)**	**1x(8+1)**
Ersatz-FUs	34.974	17.487	8.744
Schaltnetzwerk	2.401	2.291	1.587
RLB-Controller		0.783	
RLB-Speicher	0.023	0.023	0.012
Summe Hardware-Mehraufwand	38.182	20.585	11.125
Auswirkung auf den Systemtakt			
max. Frequenz ohne Redundanz (MHz)		804.51	
max. Frequenz (MHz)	758.15	769.23	762.78
Reduktion der Frequenz (%)	5.76	4.38	5.19
Ablaufzeiten des SBST			
t_{start} (kein Fehler) in Takten		414 032	
t_{start} (obere Grenze) in Takten	621 064	517 548	465 790
$t_{im\ Feld}$ (kein Fehler) in Takten		414 196	
$t_{im\ Feld}$ (obere Grenze) in Takten	621 228	517 712	465 954
$t_{im\ Feld}$ (obere Grenze) bei Systemtakt (msec)	0.819	0.673	0.611

Tabelle 6.6: Hardware-Mehraufwand, Auswirkungen auf den maximalen Systemtakt und Ablaufzeiten des SBST für die jeweils optimale RLB-Konfiguration aus Tabelle 6.5

Der Einfluss der jeweiligen RLB-Konfiguration auf die statische und dynamische Verlustleistung ist in Tabelle 6.7 zusammengefasst. Dabei enthalten die ersten drei Zeilen den prozentualen Anstieg der Verlustleistung für die zusätzlichen Komponenten. Die Verlustleistung für den RLB-Speicher wurde in dieser Tabelle dem RLB-Controller zugerechnet. In Zeile 4 ist der prozentuale Anstieg der statischen und dynamischen Verlustleistung separat zusammengefasst. Der prozentuale Anstieg der gesamten Verlustleistung der realisierten Implementierung der jeweiligen RLB-Konfigurationen befindet sich in Zeile 5. Die letzte Zeile zeigt den theoretischen Anstieg der Verlustleistung für den Fall, dass nicht verwendete funktionale Einheiten von der Versorgungsspannung abgekoppelt werden und somit keine statische Verlustleistung für diese Komponenten ensteht.

Die Ergebnisse für die zusätzliche Verlustleistung wurden durch eine Simulation des softwarebasierten Selbsttests mit Modelsim[4] zur Ermittlung der Aktivitätsdaten und anschließender Verlustleistungsanalyse durch den Encounter RTL Compiler ermittelt.

Anstieg der Verlustleistung in %						
	2x(4+2)		**2x(4+1)**		**1x(8+1)**	
	statisch	dynamisch	statisch	dynamisch	statisch	dynamisch
Ersatz-FUs	39.24	0	19.24	0	10.06	0
Schaltnetzwerk	4.48	0.19	3.32	0.21	3.31	0.21
RLB-Controller	0.60	2.21	0.60	2.21	0.60	2.21
$\sum$ stat./dyn.	44.32	2.41	23.16	2.42	13.96	2.42
Verlustleistung	12.15		7.24		5.11	
V_{DD}-gating	3.03		2.77		2.77	

Tabelle 6.7: Einfluss der jeweiligen RLB-Konfigurationen auf die Verlustleistung der VLIW-Konfiguration

[4]Simulationswerkzeug der Firma Mentor Graphics (Version SE-64 6.3c)

6.4.2 Maximierung der mittleren Lebensdauer

In diesem Abschnitt sind die Ergebnisse der Entwurfsraumexploration zur Maximierung der mittleren Lebensdauer für die gewählte Konfiguration des VLIW-Prozessors dargestellt. Die Ergebnisse in diesem Abschnitt wurden in gleicher Weise ermittelt wie im vorherigen Abschnitt erläutert.

Tabelle 6.8 enthält neben der Bezeichnung einer bestimmten RLB-Konfiguration in Spalte 1 und des resultierenden Hardware-Mehraufwandes in Spalte 2 auch den in Abschnitt 5.1.2 beschriebenen *lifetime improvement factor* (LIF) in der dritten Spalte.

Lebensdauersteigerung der VLIW-Konfiguration		
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	$LIF = \frac{MTTF_{System}}{MTTF_O}$
1x(8+1)	11.125	1.58292
1x(8+2)	20.585	1.95260[1]
1x(8+3)	30.044	2.17972
1x(8+4)	39.503	2.31599
1x(8+5)	48.962	2.39321[2]
1x(8+6)	58.422	2.43148
1x(8+7)	67.881	2.44382
1x(8+8)	77.340	2.43880
2x(4+1)	19.924	1.75472
2x(4+2)	38.182	2.16811
2x(4+3)	56.440	2.37717
2x(4+4)	74.698	2.47347
2x(4+5)	92.956	2.50675[3]
2x(4+6)	111.214	2.50429
2x(4+7)	129.472	2.48144
4x(2+1)	37.521	1.91288
4x(2+2)	73.377	2.32503
4x(2+3)	109.232	2.48101
4x(2+4)	145.088	**2.51899**
4x(2+5)	180.943	2.50196
4x(2+6)	216.798	2.45921
8x(1+1)	72.716	2.02922
8x(1+2)	143.766	2.39138
8x(1+3)	214.817	2.46927
8x(1+4)	285.867	2.44045

[Zahl] Optimale RLB-Konfiguration (maximal 500% Hardware-Mehraufwand)
[1] Optimale RLB-Konfiguration bei maximal 25% HW-Mehraufwand
[2] Optimale RLB-Konfiguration bei maximal 50% HW-Mehraufwand
[3] Optimale RLB-Konfiguration bei maximal 100% HW-Mehraufwand

Tabelle 6.8: Steigerung der mittleren Lebensdauer (LIF) verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche maximal zulässige Hardware-Mehraufwände

Auch für diese Betrachtung wurde ein maximal zulässiger Hardware-Mehraufwand von 500% festgesetzt. Sämtliche RLB-Konfigurationen innerhalb der 500% die nicht in der Tabelle aufgelistet sind, resultieren für dieses Beispiel in einem geringeren LIF und wurden aus Gründen der Übersichtlichkeit entfernt. Die optimale RLB-Konfiguration ist wieder **fett** dargestellt. Zusätzlich wurden 3 weitere Grenzen für den Hardware-Mehraufwand angenommen (25%,50% und 100%). Die optimale RLB-Konfiguration für diese Grenzen sind in der Tabelle mit Fußnoten beschriftet.

In Tabelle 6.9 sind die Auswirkungen der hervorgehobenen RLB-Konfigurationen aus Tabelle 6.8 in Bezug auf den Hardware-Mehraufwand, den Systemtakt und die Ablaufzeiten des SBST dargestellt.

Hardware-Mehraufwand in %				
	1x(8+2)	**1x(8+5)**	**2x(4+5)**	**4x(2+4)**
Ersatz-FUs	17.487	43.718	87.435	139.897
Schaltnetzwerk	2.291	4.404	4.679	4.361
RLB-Controller	0.783			
RLB-Speicher	0.023	0.058	0.058	0.046
Summe Hardware-Mehraufwand	20.585	48.962	92.956	145.088
Auswirkung auf den Systemtakt				
max. Frequenz ohne Redundanz (MHz)	804.51			
max. Frequenz (MHz)	753.58	724.64	733.14	732.06
Reduktion der Frequenz (%)	6.33	9.93	9.87	9.00
Ablaufzeiten des SBST				
t_{start} (kein Fehler) in Takten	414 032			
t_{start} (obere Grenze) in Takten	517 548	672 822	931 612	1 242 160
$t_{im\ Feld}$ (kein Fehler) in Takten	414 196			
$t_{im\ Feld}$ (obere Grenze) in Takten	517 712	672 986	931 776	1 242 324
$t_{im\ Feld}$ (obere Grenze) bei Systemtakt (msec)	0.687	0.929	1.271	1.697

Tabelle 6.9: Hardware-Mehraufwand, Auswirkungen auf den maximalen Systemtakt und Ablaufzeiten des SBST für die jeweils optimale RLB-Konfiguration aus Tabelle 6.8

Die Auswirkungen der hervorgehobenen RLB-Konfigurationen aus Tabelle 6.8 auf die zusätzliche Verlustleistung des Prozessors sind in Tabelle 6.10 dargestellt.

Anstieg der Verlustleistung in %								
	1x(8+2)		**1x(8+5)**		**2x(4+5)**		**4x(2+4)**	
	stat.	dyn.	stat.	dyn.	stat.	dyn.	stat.	dyn.
Ersatz-FUs	21.39	0	50.26	0	92.74	0	126.64	0
Schaltnetzwerk	4.45	0.21	8.33	0.24	8.42	0.23	7.98	0.22
RLB-Controller	0.60	2.21	0.60	2.21	0.60	2.21	0.60	2.21
$\sum$ stat./dyn.	26.44	2.42	59.19	2.46	101.77	2.44	135.22	2.43
Verlustleistung	8.01		15.65		25.54		33.31	
V_{DD}-gating	3.03		3.97		3.97		3.86	

Tabelle 6.10: Einfluss der jeweiligen RLB-Konfigurationen auf die Verlustleistung der VLIW-Konfiguration

6.4.3 Maximierung der Produktionsausbeute

Als ein weiteres mögliches Ziel einer Optimierung kommt die Maximierung der Produktionsausbeute infrage. Der in Abschnitt 5.1.3 beschriebene *yield improvement factor* (YIF) dient hier als Metrik. Die Ergebnisse für die gewählte VLIW-Konfiguration finden sich in Tabelle 6.11. In den ersten beiden Spalten der Tabelle befindet sich eine jeweilige RLB-Konfiguration und der resultierende Hardware-Mehraufwand. In den Spalten 3 bis 6 steht der zugehörige YIF für eine gegebene Produktionsausbeute (25%, 50%, 75% und 95%) der VLIW-Konfiguration ohne Redundanz. Für Werte des YIF größer 1 wird eine Steigerung der Produktionsausbeute erreicht. Wie bereits bei der Maximierung von Zuverlässigkeit und mittlerer Lebensdauer wurde die obere Grenze für den Hardware-Mehraufwand auf 500% festgesetzt.

Ausbeutesteigerung der VLIW-Konfiguration					
RLB-Konfiguration P**x**$(\frac{M}{P}+\frac{N}{P})$	**extra Hardware (%)**	YIF **für Ausbeuten von**			
		25%	**50%**	**75%**	**95%**
1x(8+1)	11.125	1.655	**1.296**	**1.069**	**0.930**
1x(8+2)	20.585	**1.873**	1.286	1.000	0.857
1x(8+3)	30.044	1.842	1.203	0.926	0.794
2x(4+1)	19.924	1.695	1.244	0.998	0.862
2x(4+2)	38.182	1.717	1.131	0.872	0.747
2x(4+3)	56.440	1.551	0.998	0.768	0.659
4x(2+1)	37.521	1.577	1.106	0.872	0.751
4x(2+2)	73.377	1.389	0.901	0.694	0.595
8x(1+1)	72.716	1.297	0.886	0.694	0.597
8x(1+2)	143.766	0.980	0.635	0.491	0.422

Zahl - Optimale RLB-Konfiguration

Tabelle 6.11: Steigerung der Produktionsausbeute (YIF) für verschiedene RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche Produktionsausbeuten der Konfiguration ohne Redundanz

Die Auswirkungen der optimalen RLB-Konfigurationen auf den Hardware-Mehraufwand, den Systemtakt, die Ablaufzeiten des SBST und die zusätzliche Verlustleistung wurden bereits in den Abschnitten 6.4.1 beziehungsweise 6.4.2 dargestellt und werden aus diesem Grund an dieser Stelle nicht wiederholt.

6.4.4 Auswertung der Ergebnisse

Die dargestellten Ergebnisse für die gewählte Konfiguration des VLIW-Prozessors im vorherigen Abschnitt dienten der Veranschaulichung des vorgestellten Verfahrens zur Kompensation von permanenten Fehlern und der effizienten Redundanzauswahl für Prozessorkomponenten. Im Anhang B sind die Ergebnisse für zwei weitere Konfigurationen der verwendeten VLIW-Architektur dargestellt.

Zuverlässigkeit

Im ersten Teil der Entwurfsraumexploration in Abschnitt 6.4.1 sollte die gegebene Konfiguration des VLIW-Prozessors durch den Einsatz der vorgestellten RLBs für die funktionalen Einheiten so optimiert werden, dass eine festgelegte Zuverlässigkeit erreicht wird, die im Verhältnis zur Zuverlässigkeit des ursprünglichen Systems die größte Steigerung bedeutet. Exemplarisch wurden dafür die zu erreichenden Zuverlässigkeiten von 0.9, 0.99 und 0.999 gewählt.

Dabei wurde ersichtlich, dass diese Entwurfsraumexploration für verschiedene RLB-Konfiguration nur sinnvoll ist, wenn die zu erreichende Zuverlässigkeit nicht gegen 1 tendiert. Denn für sehr hohe zu erreichende Zuverlässigkeiten eines Systems ist im wesentlichen der zusätzliche nicht redundante Hardware-Mehraufwand von entscheidender Bedeutung. Je kleiner dieser Aufwand ist, desto größer ist der positive Einfluss der mit Redundanz ausgelegten Komponenten des Systems. Diese Entwicklung geht aus der vorgestellten Modellierung der Zuverlässigkeit eines Systems in Abschnitt 5.1.1 hervor und wird durch die Ergebnisse in Tabelle 6.5 verdeutlicht. Der geringste nicht redundante Hardware-Mehraufwand für die Beispielkonfiguration des VLIW-Prozessors von ca. 11.1% ergibt sich aus der RLB-Konfiguration 1x(8+1), bestehend aus einem Segment und einem Ersatz-Logik-Block für alle 8 funktionalen Einheiten.

Bei der Verwendung der RLB-Konfiguration mit dem geringsten Hardware-Mehraufwand entsteht auch der geringste Anstieg der Verlustleistung von ca. 5% für den Fall, dass keine Abkoppelung unbenutzter Ressourcen stattfindet. Wird das vorgeschlagene V_{DD}-gating umgesetzt, erhöht sich die gesamte Verlustleistung für die drei im Detail untersuchten RLB-Konfigurationen lediglich um ca. 3%.

In Tabelle 6.6 sind zusätzlich zum Hardware-Mehraufwand auch die Auswirkungen der RLB-Konfigurationen auf den Systemtakt dargestellt. Die geringsten Auswirkungen auf diesen wird durch die Konfiguration 2x(4+1) bestehend aus 2 Segmenten und jeweils einem ELB erreicht. Für diesen Fall ergeben sich

mehr Freiheiten beim Layout, was zu kürzeren Verbindungsleitungen führt. Im gewählten Beispiel liegt die Reduktion der Taktfrequenz der untersuchten RLB-Konfigurationen zwischen 4.38% und 5.76%. Auch die Ablaufzeiten des SBST für die drei im Detail untersuchten RLB-Konfigurationen variieren lediglich zwischen ca. 0.6 und 0.8 Millisekunden.

In diesem Fall sind die Variationen der Auswirkungen der untersuchten RLB-Konfigurationen auf den Systemtakt, die Verlustleistung (mit V_{DD}-gating) und die Ablaufzeiten des SBST sehr gering. Hier ist der Anstieg der zusätzlichen Hardware der entscheidende Kostenfaktor.

Mittlere Lebensdauer

Bei der Maximierung der mittleren Lebensdauer für die gewählte Konfiguration des VLIW-Prozessors zeigte sich, dass RLB-Konfigurationen mit mehreren ELBs und Segmenten ein besseres Ergebnis bewirken und damit zu einer größeren Steigerung der mittleren Lebensdauer führen. Demzufolge ist ein minimaler nicht redundante Hardware-Mehraufwand hier nicht derart relevant wie beim Erreichen einer hohen Zuverlässigkeit. Allerdings schränkt dieser Mehraufwand, bestehend aus dem Schaltnetzwerk, dem RLB-Controller und dem Teil der LBs/ELBs in denen Fehler durch den Test nicht erkannt werden, eine beliebige Erweiterung der RLB-Konfiguration mit weiteren ELBs ein. Diese Entwicklung wurde bereits in Abschnitt 5.1.1 erläutert und wird durch die Ergebnisse in Tabelle 6.8 verdeutlicht. Für jede Art der Segmentierung ist zu erkennen, dass die Erhöhung der Anzahl der ELBs anfangs zu einer Steigerung des LIF führt. Ab einer bestimmten Anzahl von ELBs ändert sich diese Entwicklung und der LIF nimmt wieder ab. Die Maximierung der mittleren Lebensdauer um den Faktor ca. 2.52 wird für das gewählte Beispiel durch RLB-Konfiguration 4x(2+4) bestehend aus 4 Segmenten mit jeweils 4 ELBs erreicht. Der resultierende Hardware-Mehraufwand von ca. 145% dieser Konfiguration wird durch die insgesamt 16 ELBs mit ca. 140% dominiert. Die große Anzahl der ELBs dieser Konfiguration führt in diesem Fall zu einer Reduktion der maximalen Taktfrequenz von ca. 9%. Der Einfluss dieser RLB-Konfiguration sorgt für eine zusätzliche Verlustleistung von ca. 33%, falls die nicht verwendeten Komponenten nicht von der Versorgungsspannung abgekoppelt werden. Wird das vorgeschlagene V_{DD}-gating realisiert, erhöht sich die Verlustleistung für diese RLB-Konfiguration lediglich um ca. 3.9%. Die maximale Ablaufzeit des SBST ist abhängig von der Anzahl der verwendeten ELBs. Somit resultiert für die optimale RLB-Konfiguration (4x(2+4)) auch die längste mögliche Ablaufzeit des SBST von ca. 1.7 Millisekunden.

Wird eine Entwurfsraumexploration zur Ermittlung einer optimalen RLB-Konfiguration zur Maximierung der mittleren Lebensdauer verwendet, können die Kosten für den zusätzlichen Hardware-Mehraufwand enorm steigen, da diese Maximierung durch die Verwendung von mehreren ELBs erreicht wird. Eine große zusätzlich benötigte Chipfläche erhöht ebenfalls den Aufwand für das routen der Signale von den LBs zu den ELBs und umgekehrt, was eine stärkere Reduktion des Arbeitstaktes zur Folge hat. Auch die *worst-case*-Ablaufzeiten des verwendeten SBST steigen mit der Anzahl der verwendeten redundanten Komponenten. Lediglich die zusätzliche Verlustleistung bleibt für unterschiedliche RLB-Konfigurationen konstant, wenn eine Abkoppelung von der Versorgungsspannung umgesetzt wird.

Produktionsausbeute

Die Optimierung der RLB-Konfiguration zum Zwecke der Steigerung der Produktionsausbeute ist in erster Linie abhängig von der erreichbaren Ausbeute des Fertigungsprozesses und dem notwendigen Hardware-Mehraufwand einer RLB-Konfiguration. Für die gewählte Konfiguration des VLIW-Prozessors ist in Tabelle 6.11 zu sehen, dass bei einer Produktionsausbeute des Prozessors ohne Redundanz von 95% für alle RLB-Konfigurationen ein $YIF < 1$ erreicht wird. Somit kann durch keine RLB-Konfiguration eine Steigerung der funktionierenden Chips pro Wafer erreicht werden. Weiterhin ist auch zu sehen, dass für diese VLIW-Konfiguration bei Produktionsausbeuten bis einschließlich 75% ein $YIF > 1$ erreicht werden kann. Dadurch wird grundsätzlich auch eine Steigerung der Produktionsausbeute möglich. Allerdings ist hier ein Auswahlprozess für verschiedene RLB-Konfigurationen nur notwendig, wenn die ursprüngliche Produktionsausbeute sehr niedrig ist.

KAPITEL 7

Zusammenfassung und Ausblick

Die vorliegende Arbeit beschäftigt sich mit der effizienten Auswahl von Redundanz für beliebige Komponenten von Prozessoren zur Kompensation von permanenten Fehlern. Dafür wurde eine grundlegende Architektur vorgestellt, mit der ein dauerhaftes Fehlverhalten im weiteren Betrieb verhindert wird. Dabei wurde die Umsetzung so gestaltet, dass dieses Verfahren unabhängig von der eigentlichen Fehlerursache angewendet werden kann und damit zur Kompensation von Produktionsfehlern, Alterungsfehlern und zufällige permanente Fehler zur Verfügung steht. Die vorgestellte Architektur erweitert ausgewählte Prozessorkomponenten auf Registertransferebene mit zuschaltbarer Redundanz. Diese Implementierung orientiert sich an bereits bestehenden Verfahren zur Kompensation von permanenten Fehlern in Speichern und FPGAs. Das vorgestellte Verfahren lässt sich in die Kategorie aktive Hardware-Redundanz einordnen und kompensiert permanente Fehler durch eine Kombination aus Test, Diagnose und Rekonfiguration.

Im Gegensatz zu anderen Arbeiten, die eine Implementierung von Hardware-Redundanz zur Kompensation von permanenten Fehler leisten, beschäftigt sich diese Arbeit im Detail mit einer effizienten und damit zielgerichteten Auswahl von Redundanz. Damit unterscheidet sich die Herangehensweise dieser Arbeit grundlegend von anderen, weil durch entsprechende Zielvorgaben eine optimale Auswahl von vorhandenen Komponenten einschließlich der notwendigen Menge an Redundanz stattfindet. Somit ermöglicht diese Arbeit eine Optimierung der Zuverlässigkeit, Lebensdauer oder auch Produktionsausbeute im Entwurfsprozess von integrierten Schaltungen, welche durch die steigende Anfälligkeit sinkender Strukturgrößen auch gegenüber permanenten Fehlern notwendig wird.

Die vorgestellte Optimierung dieser Arbeit ist vor allem für anwendungsspezifische integrierte Schaltungen interessant, bei denen bereits eine Optimierung im Bezug auf eine möglichst hohe parallele Abarbeitung eines Algorithmus durchgeführt wurde. Solche anwendungsspezifischen Prozessoren verfügen dann in der

Regel bereits über eine bestimmte Homogenität, welche beim Einsatz von zuschaltbarer Redundanz weniger Kosten für zusätzliche Hardware verursacht und damit zu einer höheren Effizienz im Gegensatz zur Verwendung von Hardware-Redundanz in heterogenen Systemen führt. Dadurch wird es möglich, zusätzlich zur anwendungsorientierten Optimierung von Prozessoren, einen weiteren Optimierungsschritt umzusetzen, welcher beispielsweise eine Maximierung einer bestimmten Einsatzzeit ermöglicht und gleichzeitig die geforderte Funktionalität bereitstellt. An dieser Stelle wird auch deutlich, dass eine mögliche Degradation im Fehlerfall nicht in jedem Fall eine Lösung ist.

Die Auswahl der Prozessorkomponenten auf Register-Transfer und nicht auf Logik-Ebene ergibt sich aus folgenden Gründen: In vorangegangenen Arbeiten wurde bereits eine Extraktion identischer Teilschaltungen aus gegebenen Logik-Netzlisten untersucht [1, 7]. Durch die Modellierung des Systems in dieser Arbeit zeigte sich allerdings, dass die entstehenden Teilnetze aufgrund ihres Aufbaus und dem damit verbundenen hohen Hardware-Mehraufwand nicht geeignet sind, um durch die vorgestellte Redundanz erweitert zu werden. Weiterhin wäre eine Integration dieser Strategie für kleinere Teilschaltungen in einen bestehenden Entwurfsprozess wesentlich aufwendiger umzusetzen. Zusätzlich ergeben sich Probleme beim Test und einer notwendigen fein-granularen Diagnose. Der Einsatz von Redundanz für die funktionalen Einheiten des VLIW-Prozessors ist aufgrund der hohen Temperaturabhängigkeit von Alterungsfehlern (siehe Abschnitt 2.3.2) und der hohen Temperaturentwicklung in diesen Komponenten [MLN+06] besonders geeignet.

Im Allgemeinen ist eine fein-granulare Umsetzung des vorgestellten Verfahrens auch aufgrund der großen zusätzlichen Verzögerungen durch die Schaltnetzwerke und die damit einhergehenden Auswirkungen auf den Systemtakt als ungünstig zu bewerten. Gerade diese Tatsache macht die Notwendigkeit eines optimierten Verfahrens zur Auswahl von Hardware-Redundanz auf RT-Ebene deutlich, welches durch die vorliegende Arbeit bereitgestellt wird.

Möglichkeiten zur Erweiterung der vorgestellten Architektur ergeben sich in folgenden Bereichen: Durch die Tatsache das vor allem bei Alterungsfehlern eine enorme Temperaturabhängigkeit besteht, sind Mechanismen zur Lastverteilung im Feld eine sinnvolle Erweiterung [17], die zusätzlich zur Möglichkeit der Kompensation von permanenten Fehlern dieser Arbeit eine Abschwächung der Fehlerursache bewirken können. Als auslösendes Ereignis für einen Test mit anschließender Rekonfiguration im Fehlerfall wurde in dieser Arbeit lediglich der Systemstart und ein zyklischer Test realisiert. An dieser Stelle ergeben sich ebenfalls Möglichkeiten der Erweiterung der vorgestellten Architektur. Dazu können

beispielsweise compilerbasierte Techniken eingesetzt werden, mit deren Hilfe Abweichungen festgestellt und der Rekonfigurationsprozess gestartet wird. Typische Fehlersignale von Mechanismen der Fehlertoleranz für transiente Fehler oder auch Alterungssensoren auf dem Chip können ebenfalls als auslösendes Ereignis dienen. Die Ergebnisse des Anwendungsbeispiels dieser Arbeit zeigen eine deutliche Steigerung der Verlustleistung bei der Verwendung einer großen Anzahl an Ersatz-Logik-Blöcken von bis zu 33%. Bei einer praktischen Umsetzung muss in jedem Fall eine Abschaltung der Versorgungsspannung nicht benutzer LB/ELBs realisiert werden, was die Steigerung der Verlustleistung für verschiedene RLB-Konfigurationen auf vernachlässigbare Werte senkt.

ANHANG A

Befehlssatz der VLIW-Architektur

In Tabelle A.1 ist der Befehlssatz der implementierten generischen VLIW-Architektur abgebildet. Verschiedene Konfigurationen des VLIW-Prozessors für die Anzahl der verfügbaren Register jeder Registerbank führen daher zu unterschiedlichen Kodierungen für die Quell- und Zielregister und auch für das Immediate. Die Befehlskodierungen für nicht Immediate-Befehle ist in Tabelle A.2 und für Immediate-Befehle in Tabelle A.3 abgebildet.

Mnemonic	**Bedeutung**	**OPCODE**	**SOURCE 1**	**SOURCE 2**	**DESTINATION**
NOP	no operation	00000000	-	-	-
MOVA Rx, Rz	Rz <= Rx	00000001	Rx	-	Rz
INCA Rx, Rz	Rz <= Rx + 1	00000010	Rx	-	Rz
DECA Rx, Rz	Rz <= Rx - 1	00000011	Rx	-	Rz
ADD Rx, Ry, Rz	Rz <= Rx + Ry	00000100	Rx	Ry	Rz
SUB Rx, Ry, Rz	Rz <= Rx - Ry	00000101	Rx	Ry	Rz
MULH Rx, Ry, Rz	Rz <= high(Rx * Ry)	00000110	Rx	Ry	Rz
MULL Rx, Ry, Rz	Rz <= low(Rx * Ry)	00000111	Rx	Ry	Rz
SEQ Rx, Ry, Rz	Rz <= 00..01 if (Rx == Ry) else Rz<= 00..00	00001000	Rx	Ry	Rz
SNE Rx, Ry, Rz	Rz <= 00..01 if (Rx != Ry) else Rz<= 00..00	00001001	Rx	Ry	Rz
SGE Rx, Ry, Rz	Rz <= 00..01 if (Rx >= Ry) else Rz<= 00..00	00001010	Rx	Ry	Rz

SLE Rx, Ry, Rz	Rz <= 00..01 if (Rx =< Ry) else Rz<= 00..00	00001011	Rx	Ry	Rz
SGT Rx, Ry, Rz	Rz <= 00..01 if (Rx > Ry) else Rz<= 00..00	00001100	Rx	Ry	Rz
SGT Rx, Ry, Rz	Rz <= 00..01 if (Rx < Ry) else Rz<= 00..00	00001101	Rx	Ry	Rz
AND Rx, Ry, Rz	Rz <= Rx AND Ry	00001110	Rx	Ry	Rz
OR Rx, Ry, Rz	Rz <= Rx OR Ry	00001111	Rx	Ry	Rz
XOR Rx, Ry, Rz	Rz <= Rx XOR Ry	00010000	Rx	Ry	Rz
NOTA Rx, Rz	Rz <= NOT Rx	00010001	Rx	-	Rz
SLL Rx, Ry, Rz	Rz <= Rx sll Ry	00010010	Rx	Ry	Rz
SRL Rx, Ry, Rz	Rz <= Rx srl Ry	00010011	Rx	Ry	Rz
SRA Rx, Ry, Rz	Rz <= Rx sra Ry	00010100	Rx	Ry	Rz
JMPR Rx	PC <= Rx	00010101	Rx	-	-
JMPZ Rx, Ry	PC <= Rx if (Ry == 0)	00010110	Rx	Ry	-
JMPNZ Rx, Ry	PC <= Rx if (Ry != 0)	00010111	Rx	Ry	-
LOAD Rx, Rz	Rz <= MEM[Rx]	00011000	Rx	-	Rz
STORE Rx, Ry	MEM[Rx] <= Ry	00011001	Rx	Ry	-
CALL Rx, Rz	PC <= Rx; Rz <= PC	00011010	Rx	-	Rz
Mnemonic	**Bedeutung**	**OPCODE**	**IMMEDIATE**		**DESTINATION**
LDC <VALUE>, Rz	Rz <= <VALUE>	1111	<VALUE>		Rz

Tabelle A.1: Befehlssatz der generischen VLIW-Architektur

OPCODE	SOURCE 1	SOURCE 2	DESTINATION
(3R+7 downto 3R)	(3R-1 downto 2R)	(2R-1 downto R)	(R-1 downto 0)

Tabelle A.2: Kodierung für nicht Immediate-Befehle

OPCODE	IMMEDIATE	DESTINATION
(3R+7 downto 3R+4)	(3R+3 downto R)	(R-1 downto 0)

Tabelle A.3: Kodierung für Immediate-Befehle

ANHANG B

Anhang

B.1 Ergebnisse für VLIW-Konfiguration

$C = 1; F = 2; W = 4; R = 4$

In den folgenden Tabellen befinden sich die Ergebnisse für eine VLIW-Konfiguration bestehend aus einem Cluster $C = 1$, zwei Ausführungspfaden $F = 2$, einer Datenbreite von 16-Bit $W = 4$ und 16 Registern der Registerbank $R = 4$.

In Tabelle B.1 ist die Flächenverteilung der VLIW-Konfiguration ohne Redundanz aufgelistet. Die erreichte Fehlerüberdeckung des SBST ist in Tabelle B.2 und ihr Speicherbedarf in Tabelle B.3 zu finden.

Flächenverteilung der gewählten Konfiguration des VLIW-Prozessors			
Komponente	**Anzahl**	**$\sum$ Fläche in μm^2**	**Fläche in %**
VLIW-Prozessor	1	13 443	100
Programmzähler (PC)	1	192	1.428
Speicherverwaltung	1	337	2.507
Registerbank	1	2 111	15.703
FE (Fetch)	2	307	2.284
DE (Decode)	2	2 547	18.947
EX (Execute)	2	436	3.243
FU	2	7 087	52.719
MA (Memory Access)	2	426	3.169

Tabelle B.1: Flächenverteilung für die gewählte Konfiguration

Fehlerüberdeckung des SBST		
	Fehlermodell	
	Haftfehler	**Transitionsfehler**
Testmuster	421	
alle Fehlerpunkte	15 980	10 947
erkannte Fehlerpunkte	15 404	10 920
Fehlerüberdeckung (einzeln)	96.40 %	99.75 %
Fehlerüberdeckung (insgesamt)	**97.761 %**	

Tabelle B.2: Zusammensetzung der Fehlerüberdeckung für eine funktionale Einheit mit 16-Bit Datenbreite

Speicherbedarf des SBST		
	Programmspeicher	**Datenspeicher**
LOC	318	2 105
Datenmenge (kB)	1.55	4.11
Datenmenge insgesamt (kB)	**5.66**	

Tabelle B.3: Speicherbedarf des SBST für Programm- und Datenspeicher

B.1.1 Maximierung der Zuverlässigkeit

Tabelle B.4 zeigt die mögliche Zuverlässigkeitssteigerung als *reliability improvement factor* für verschiedene RLB-Konfigurationen und unterschiedlich zu erreichende Zuverlässigkeiten. In Tabelle B.5 sind die Auswirkungen auf die Verlustleistung dargestellt. Anschließend sind die Auswirkungen der optimalen RLB-Konfiguration auf die zusätzliche Hardware, den Systemtakt und die Ablaufzeiten des SBST in Tabelle B.6 zusammengefasst.

Zuverlässigkeitssteigerung der gewählten VLIW Konfiguration				
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	RIF **für** $R_{VLIW_{RLB}} =$ 0.9	RIF **für** $R_{VLIW_{RLB}} =$ 0.99	RIF **für** $R_{VLIW_{RLB}} =$ 0.999
1x(2+1)	39.550	**1.44**	**1.52**	**1.53**
1x(2+2)	67.510	1.43	1.46	1.46
1x(2+3)	95.470	1.37	1.40	1.40
1x(2+4)	123.430	1.32	1.34	1.34
2x(1+1)	66.120	1.42	1.49	1.49
2x(1+2)	120.649	1.36	1.39	1.39

[Zahl] - RLB-Konfiguration mit maximaler Zuverlässigkeitssteigerung

Tabelle B.4: Einfluss verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche zu erreichende Zuverlässigkeiten 0.9|0.99|0.999

Anstieg der Verlustleistung in %		
	1x(2+1)	
	statisch	dynamisch
Ersatz-FUs	31.54	0
Schaltnetzwerk	4.73	1.26
RLB-Controller	6.93	6.73
$\sum$ stat./dyn.	43.20	7.99
Verlustleistung	10.95	
V_{DD}-gating	8.30	

Tabelle B.5: Einfluss der RLB-Konfiguration auf die Verlustleistung der VLIW-Konfiguration

Hardware-Mehraufwand in %	
	1x(2+1)
Ersatz-FUs	26.103
Schaltnetzwerk	3.642
RLB-Controller	9.723
RLB-Speicher	0.083
Summe Hardware-Mehraufwand	39.550
Auswirkung auf den Systemtakt	
max. Frequenz ohne Redundanz (MHz)	1 222.49
max. Frequenz (MHz)	1 165.50
Reduktion der Frequenz (%)	4.66
Ablaufzeiten des SBST	
t_{start} (kein Fehler) in Takten	24 860
t_{start} (obere Grenze) in Takten	37 294
$t_{im\ Feld}$ (kein Fehler) in Takten	24 960
$t_{im\ Feld}$ (obere Grenze) in Takten	37 394
$t_{im\ Feld}$ (obere Grenze) bei Systemtakt (msec)	0.032

Tabelle B.6: Hardware-Mehraufwand, Auswirkungen auf den maximalen Systemtakt und Ablaufzeiten des SBST für die optimale RLB-Konfiguration aus Tabelle B.4

B.1.2 Maximierung der Lebensdauer

Tabelle B.7 beinhaltet den *lifetime improvement factor* für verschiedene RLB-Konfigurationen. Die Auswirkungen auf die Verlustleistung zeigt Tabelle B.8. Anschließend ist in Tabelle B.9 der resultierende Hardware-Mehraufwand für die optimale RLB-Konfiguration detailliert aufgelistet.

Lebensdauersteigerung der VLIW-Konfiguration		
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	$LIF = \frac{MTTF_{System}}{MTTF_O}$
1x(2+1)	39.550	1.18114
1x(2+2)	67.510	1.27366
1x(2+3)	95.470	**1.29121**
1x(2+4)	123.430	1.27636
2x(1+1)	66.120	1.20431
2x(1+2)	120.649	1.26158
2x(1+3)	175.179	1.23998

[Zahl] - optimale RLB-Konfiguration

Tabelle B.7: Steigerung der mittleren Lebensdauer (LIF) verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration

Anstieg der Verlustleistung in %		
	1x(2+3)	
	statisch	dynamisch
Ersatz-FUs	87.33	0
Schaltnetzwerk	8.33	2.61
RLB-Controller	7.03	7.23
$\sum$ stat./dyn.	102.69	9.84
Verlustleistung	17.66	
V_{DD}-gating	10.30	

Tabelle B.8: Einfluss der optimalen RLB-Konfigurationen auf die Verlustleistung der VLIW-Konfiguration

Hardware-Mehraufwand in %	
	1x(2+3)
Ersatz-FUs	78.308
Schaltnetzwerk	7.190
RLB-Controller	9.723
RLB-Speicher	0.249
Summe Hardware-Mehraufwand	95.470
Auswirkung auf den Systemtakt	
max. Frequenz ohne Redundanz (MHz)	1 222.49
max. Frequenz (MHz)	1 121.08
Reduktion der Frequenz (%)	8.30
Ablaufzeiten des SBST	
t_{start} (kein Fehler) in Takten	24 860
t_{start} (obere Grenze) in Takten	62 162
$t_{im\ Feld}$ (kein Fehler) in Takten	24 960
$t_{im\ Feld}$ (obere Grenze) in Takten	62 262
$t_{im\ Feld}$ (obere Grenze) bei Systemtakt (msec)	0.056

Tabelle B.9: Hardware-Mehraufwand, Auswirkungen auf den maximalen Systemtakt und Ablaufzeiten des SBST für die optimale RLB-Konfiguration aus Tabelle B.7

B.1.3 Maximierung der Produktionsausbeute

Die Auswirkungen auf die Produktionsausbeute sind in Tabelle B.10 dargestellt.

Ausbeutesteigerung der VLIW-Konfiguration					
RLB-Konfiguration P**x**$(\frac{M}{P}+\frac{N}{P})$	**extra Hardware (%)**	YIF **für Ausbeuten von**			
		25%	**50%**	**75%**	**95%**
1x(2+1)	39.550	0.924	0.851	0.781	0.729
1x(2+2)	67.510	0.854	0.733	0.653	0.606
2x(1+1)	66.120	0.797	0.719	0.655	0.612
2x(1+2)	120.649	0.636	0.546	0.491	0.459

Tabelle B.10: Steigerung der Produktionsausbeute (YIF) für verschiedene RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche Produktionsausbeuten der Konfiguration ohne Redundanz

B.2 Ergebnisse für VLIW-Konfiguration $C = 2; F = 8; W = 7; R = 6$

In den folgenden Tabellen befinden sich die Ergebnisse für eine VLIW-Konfiguration bestehend aus zwei Clustern $C = 2$, 8 Ausführungspfaden $F = 8$, einer Datenbreite von 128-Bit $W = 7$ und 64 Registern pro Registerbank $R = 6$.

In Tabelle B.11 ist die Flächenverteilung der VLIW-Konfiguration ohne Redundanz aufgelistet. Die erreichte Fehlerüberdeckung des SBST ist in Tabelle B.12 und ihr Speicherbedarf in Tabelle B.13 zu finden.

Flächenverteilung der gewählten Konfiguration des VLIW-Prozessors			
Komponente	**Anzahl**	**$\sum$ Fläche in μm^2**	**Fläche in %**
VLIW-Prozessor	1	3 091 708	100
Programmzähler (PC)	1	3 457	0.112
Speicherverwaltung	1	28 330	0.916
Registerbank	2	233 799	7.562
FE (Fetch)	16	3 130	0.101
DE (Decode)	16	404 403	13.080
EX (Execute)	16	23 767	0.769
FU	16	2 371 135	76.693
MA (Memory Access)	16	23 687	0.766

Tabelle B.11: Flächenverteilung für die gewählte Konfiguration

Fehlerüberdeckung des SBST		
	Fehlermodell	
	Haftfehler	**Transitionsfehler**
Testmuster	2 534	
alle Fehlerpunkte	449 278	677 490
erkannte Fehlerpunkte	447 519	670 025
Fehlerüberdeckung (einzeln)	99.62 %	98.90 %
Fehlerüberdeckung (insgesamt)	**99.18 %**	

Tabelle B.12: Zusammensetzung der Fehlerüberdeckung für eine funktionale Einheit mit 128-Bit Datenbreite

Speicherbedarf des SBST		
	Programmspeicher	**Datenspeicher**
LOC	2 036	12 670
Datenmenge (kB)	105.87	202.72
Datenmenge insgesamt (kB)	**308.59**	

Tabelle B.13: Speicherbedarf des SBST für Programm- und Datenspeicher

B.2.1 Maximierung der Zuverlässigkeit

Tabelle B.14 zeigt die mögliche Zuverlässigkeitssteigerung als *reliability improvement factor* für verschiedene RLB-Konfigurationen und unterschiedlich zu erreichenden Zuverlässigkeiten. Anschließend sind die Auswirkungen der optimalen RLB-Konfiguration auf die zusätzliche Hardware in Tabelle B.15 zusammengefasst.

Zuverlässigkeitssteigerung der gewählten VLIW-Konfiguration				
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	RIF **für** $R_{VLIW_{RLB}} =$ 0.9	RIF **für** $R_{VLIW_{RLB}} =$ 0.99	RIF **für** $R_{VLIW_{RLB}} =$ 0.999
1x(16+1)	5.672	2.67	3.65	**3.83**
1x(16+2)	10.744	3.20	**3.74**	3.79
1x(16+3)	15.816	3.24	3.69	3.74
1x(16+4)	20.888	3.22	3.64	3.69
2x(8+1)	10.423	2.89	3.71	3.83
2x(8+2)	20.246	**3.25**	3.73	3.78
2x(8+3)	30.068	3.24	3.67	3.71
4x(4+1)	19.925	3.03	3.73	3.82
4x(4+2)	39.249	3.25	3.70	3.74
4x(4+3)	58.573	3.21	3.62	3.66
8x(2+1)	38.928	3.11	3.71	3.78
8x(2+2)	77.255	3.21	3.63	3.67
8x(2+3)	115.582	3.14	3.53	3.57
16x(1+1)	76.934	3.12	3.66	3.72
16x(1+2)	153.268	3.12	3.51	3.55
16x(1+3)	229.601	3.01	3.36	3.39

[Zahl] - RLB-Konfiguration mit maximaler Zuverlässigkeitssteigerung

Tabelle B.14: Einfluss verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche zu erreichende Zuverlässigkeiten 0.9|0.99|0.999

Hardware-Mehraufwand in %			
	2x(8+2)	**1x(16+2)**	**1x(16+1)**
Ersatz-FUs	18.950	9.475	4.737
Schaltnetzwerk	1.119	1.092	0.760
RLB-Controller	0.171		
RLB-Speicher	0.006	0.006	0.003
Summe Hardware-Mehraufwand	20.246	10.744	5.672

Tabelle B.15: Hardware-Mehraufwand für die jeweils optimale RLB-Konfiguration aus Tabelle B.14

B.2.2 Maximierung der Lebensdauer

Tabelle B.16 beinhaltet den *lifetime improvement factor* für verschiedene RLB-Konfigurationen. Anschließend ist in Tabelle B.17 der resultierende Hardware-Mehraufwand für die optimalen RLB-Konfiguration detailliert aufgelistet.

Lebensdauersteigerung der VLIW-Konfiguration		
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{K}{P}$**)**	**extra Hardware (%)**	$LIF = \frac{MTTF_{System}}{MTTF_O}$
1x(16+1)	5.672	1.68282
1x(16+2)	10.744	2.16892
1x(16+3)	15.816	2.51176
1x(16+4)	20.888	2.75230[1]
1x(16+5)	25.960	2.91917
1x(16+6)	31.032	3.03259
1x(16+7)	36.104	3.10700
1x(16+8)	41.176	3.15281
1x(16+9)	46.248	3.17758
1x(16+10)	51.320	3.17684
2x(8+1)	10.423	1.91643
2x(8+2)	20.246	2.50804
2x(8+3)	30.068	2.86835
2x(8+4)	39.891	3.08087
2x(8+5)	49.714	3.20030[2]
2x(8+6)	59.537	3.26099
2x(8+7)	69.359	3.28439
2x(8+8)	79.182	3.28404
4x(4+1)	19.925	2.16220
4x(4+2)	39.249	2.81091
4x(4+3)	58.573	3.12953
4x(4+4)	77.897	3.27114
4x(4+5)	97.222	3.31986[3]
4x(4+6)	116.546	3.31964
4x(4+7)	135.870	3.29351
8x(2+1)	38.928	2.38497
8x(2+2)	77.255	3.01947
8x(2+3)	115.582	3.24657
8x(2+4)	153.910	3.29751
8x(2+5)	192.237	3.27397
16x(1+1)	76.934	2.54673
16x(1+2)	153.268	3.09508
16x(1+3)	229.601	3.20076
16x(1+4)	305.935	3.15614

[1] Optimale RLB-Konfiguration bei maximal 25% HW-Mehraufwand
[2] Optimale RLB-Konfiguration bei maximal 50% HW-Mehraufwand
[3] Optimale RLB-Konfiguration bei maximal 100% HW-Mehraufwand

Tabelle B.16: Steigerung der mittleren Lebensdauer (LIF) verschiedener RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche maximal zulässige Hardware-Mehraufwände

Hardware-Mehraufwand in %			
	1x(16+4)	**2x(8+5)**	**4x(4+5)**
Ersatz-FUs	18.950	47.374	94.748
Schaltnetzwerk	1.755	2.154	2.288
RLB-Controller	0.171		
RLB-Speicher	0.012	0.014	0.014
Summe Hardware-Mehraufwand	20.888	49.714	97.222

Tabelle B.17: Hardware-Mehraufwand für die jeweils optimale RLB-Konfiguration aus Tabelle B.16

B.2.3 Maximierung der Produktionsausbeute

Die Auswirkungen auf die Produktionsausbeute sind in Tabelle B.18 dargestellt.

Ausbeutesteigerung der VLIW-Konfiguration					
RLB-Konfiguration P**x(**$\frac{M}{P}+\frac{N}{P}$**)**	**extra Hardware (%)**	YIF **für Ausbeuten von**			
		25%	**50%**	**75%**	**95%**
1x(16+1)	5.672	1.872	1.420	**1.145**	0.982
1x(16+2)	10.744	2.253	**1.475**	1.113	0.936
1x(16+3)	15.816	**2.313**	1.428	1.064	0.895
1x(16+4)	20.888	2.252	1.367	1.018	0.856
2x(8+1)	10.423	2.015	1.416	1.106	0.939
2x(8+2)	20.246	2.202	1.372	1.024	0.861
2x(8+3)	30.068	2.096	1.271	0.945	0.795
4x(4+1)	19.925	2.010	1.336	1.022	0.864
4x(4+2)	39.249	1.941	1.185	0.883	0.742
8x(2+1)	38.928	1.814	1.165	0.882	0.744
8x(2+2)	77.255	1.524	0.926	0.690	0.581
16x(1+1)	76.934	1.450	0.915	0.690	0.582
16x(1+2)	153.268	1.049	0.639	0.478	0.403

[Zahl] - Optimale RLB-Konfiguration

Tabelle B.18: Steigerung der Produktionsausbeute (YIF) für verschiedene RLB-Konfigurationen der gewählten VLIW-Konfiguration für unterschiedliche Produktionsausbeuten der Konfiguration ohne Redundanz

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ALU ..... Arithmetic Logic Unit
AR ..... Array Redundancy
ATE ..... Automatic Test Equipment
ATPG ..... Automatic Test Pattern Generation
BDD ..... Binary Decision Diagram
BISR ..... Built-In Self-Repair
BIST ..... Built-In Self-Test
CISC ..... Complex Instruction Set Computer
CLR ..... Component Level Redundancy
CMOS ..... Complementary Metal-Oxid-Semiconductor
CPU ..... Central Processing Unit
DFT ..... Dynamic Fault Tree
DIVA ..... Dynamic Implementation and Verification Architecture
DPL ..... Double Pass transistor Logic
DQR ..... Dynamic Queue Redundancy
DRBD ..... Dynamic Reliability Block Diagram
ECC ..... Error Correction Code
EDAC ..... Error Detecting And Correcting
EDC ..... Error Detecting Code
ELB ..... Ersatz-Logik-Block
ELF ..... Early Life Failure
EM ..... Electro Migration
FPGA ..... Field-Programmable Gate Array
FPU ..... Floating-Point Unit
FT ..... Fault Tree
FU ..... Functional Unit
HBD ..... Hard-BreakDown
HCI ..... Hot-Carrier Injection

HYETI ..... High Yield and Error Tolerant Integration
IC ..... Integrated Circuit
ITRS ..... International Technology Roadmap for Semiconductors
LB ..... Logik-Block
LBIST ..... Logic Built-In Self-Test
LFSR ..... Linear Feedback Shift Register
LOC ..... Lines Of Code
MEU ..... Multiple Event Upsets
MISR ..... Multiple Input Signature Register
MOSFET .... Metal-Oxid-Semiconductor Field-Effect Transistor
MTBF ..... Mean Time Between Failures
MTTF ..... Mean Time To Failure
MTTR ..... Mean Time To Recover
NBTI ..... Negative-Bias Temperature Instability
NMOS ..... N-channel Metal-Oxide-Semiconductor
PAY ..... Performance Averaged Yield
PBD ..... Post-BreakDown
PLA ..... Programmable Logic Array
PMOS ..... P-channel Metal-Oxide-Semiconductor
PTL ..... Pass Transistor Logic
RAM ..... Random-Access Memory
RAMP ..... Reliability Aware Micro Processor
RAW ..... Read After Write
RBD ..... Reliability Block Diagram
RISC ..... Reduced Instruction Set Computer
RLB ..... Redundanter Logik-Block
RTL ..... Register Transfer Level
SBD ..... Soft-BreakDown
SBST ..... Software-Based Self-Test
SEB ..... Single Event Burnout
SEGR ..... Single Event Gate Rupture
SEL ..... Single Event Latchup
SET ..... Single Event Transient
SEU ..... Single Event Upset

SIHFT ..... Software-Implemented Hardware Fault Tolerance
SIMD ..... Single Instruction Multiple Data
SM ..... Stress Migration
SoC ..... System-on-a-Chip
SRAM ..... Static Random-Access Memory
TC ..... Thermal Cycling
TDDB ..... Time-Dependent Dielectric Breakdown
TMR ..... Triple Modular Redundancy
VLIW ..... Very Long Instruction Word
VLSI ..... Very-Large-Scale Integration

Literaturverzeichnis

[ABMC08] Todd Austin, Valeria Bertacco, Scott Mahlke, and Yu Cao. Reliable systems on unreliable fabrics. *IEEE Design & Test of Computers*, 25(4):322–332, 2008.

[ABMF04] Todd Austin, David Blaauw, Trevor Mudge, and Krisztián Flautner. Making typical silicon matter with razor. *Computer*, 37(3):57–65, March 2004.

[ACF+03] D. Anand, B. Cowan, O. Farnsworth, P. Jakobsen, S. Oakland, M.R. Ouellette, and D.L. Wheater. An on-chip self-repair calculation and fusing methodology. *IEEE Design & Test of Computers*, 20(5):67–75, 2003.

[Ala08] M. Alam. Reliability- and process-variation aware design of integrated circuits. *Microelectronics Reliability*, 48(8-9):1114 – 1122, 2008.

[ALR04] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 91–120. Springer US, 2004.

[AM05] M.A. Alam and S. Mahapatra. A comprehensive model of pmos nbti degradation. *Microelectronics Reliability*, 45(1):71 – 81, 2005.

[APGP07] A. Apostolakis, M. Psarakis, D. Gizopoulos, and A. Paschalis. A functional self-test approach for peripheral cores in processor-based socs. In *Proc. 13th IEEE International On-Line Testing Symposium IOLTS 2007*, pages 271–276, 2007.

[APR04] A. Agarwal, B.C. Paul, and K. Roy. A novel fault tolerant cache to improve yield in nanometer technologies. In *Proc. 10th IEEE International On-Line Testing Symposium IOLTS 2004*, pages 149–154, 2004.

[Aug12] Michael Augustin. *Spezifische Fehlertoleranz für kombinatorische und sequentielle Schaltungen*. PhD thesis, BTU Cottbus, 2012.

[Aus99] T.M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[AV10] Jaume Abella and Xavier Vera. Electromigration for microarchitects. *ACM Computing Surveys*, 42(2):1–18, 2010.

[AVU+07] J. Abella, X. Vera, O.S. Unsal, O. Ergin, and A. Gonzalez. Fuse: A technique to anticipate failures due to degradation in alus. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 15–22, 2007.

[AVU+08] J. Abella, X. Vera, O.S. Unsal, O. Ergin, A. Gonzalez, and J.W. Tschanz. Refueling: Preventing wire degradation due to electromigration. *Micro, IEEE*, 28(6):37–46, 2008.

[BBSS98] M. Broglia, G. Buonanno, M.G. Sami, and M. Selvini. Designing for yield: a defect-tolerant approach to high-level synthesis. In *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on*, pages 312–317, 1998.

[BFGM07] J. Blome, Shuguang Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 109–122, 2007.

[Bla69] James R. Black. Electromigration - a brief survey and some recent results. *Electron Devices, IEEE Transactions on*, 16(4):338–347, 1969.

[Bor05] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. 25(6):10–16, 2005.

[BSO05] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. 38th Annual IEEE/ACM International Symposium on MICRO-38 Microarchitecture*, pages 12 pp.–, 2005.

[BSOS04] F.A. Bower, P.G. Shealy, S. Ozev, and D.J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proc. International Conference on Dependable Systems and Networks*, pages 51–60, 2004.

[CD00] L. Chen and S. Dey. Defuse: a deterministic functional self-test methodology for processors. In *Proc. 18th IEEE VLSI Test Symposium*, pages 255–262, 2000.

[CMAB09] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. A flexible software-based framework for online detection of hardware defects. *IEEE Transactions on Computers*, 58(8):1063–1079, August 2009.

[Con03] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Micro, IEEE*, 23(4):14–19, 2003.

[CSRS04] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero. Automatic test program generation: a case study. *IEEE Design & Test of Computers*, 21(2):102–109, March 2004.

[CWLG07] Chung-Ho Chen, Chih-Kai Wei, Tai-Hua Lu, and Hsun-Wei Gao. Software-based self-testing with multiple-level abstractions for soft processor cores. *IEEE Trans. VLSI Syst.*, 15(5):505–517, 2007.

[DAKP07] A. Dutta, S. Alampally, A. Kumar, and R. A. Parekhji. A bist implementation framework for supporting field testability and configurability in an automotive soc. In *Proceedings of the 2007 Workshop on Dependable and Secure Nanocomputing*, 2007.

[DJ08] A. Dutta and A. Jas. Combinational logic circuit protection using customized error detecting and correcting codes. In *Proc. 9th International Symposium on Quality Electronic Design ISQED 2008*, pages 68–73, 2008.

[DNR02] E. Dupont, M. Nicolaidis, and P. Rohr. Embedded robustness ips for transient-error-free ics. *IEEE Design & Test of Computers*, 19(3):54–68, 2002.

[Dod07] Jonathan Dodge. Reduce circuit zapping from cosmic radiation. *Power Electronics Technology*, September, September 2007.

[DQ11] Carter DeHon and Quinn. Final report for ccc cross-layer reliability visioning study. Technical report, Computing Research Association (CRA) for the Computing Community Consortium (CCC), March 2011.

[DX06] S. Distefano and L. Xing. A new approach to modeling the system reliability: dynamic reliability block diagrams. In *Proc. Annual Reliability and Maintainability Symposium RAMS '06*, pages 189–195, 2006.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[HBS+04] Zhigang Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 32–37, 2004.

[HCW06] Yu-Ying Hsiao, Chao-Hsun Chen, and Cheng-Wen Wu. A built-in self-repair scheme for nor-type flash memory. In *Proc. 24th IEEE VLSI Test Symposium*, pages 6 pp.–119, 2006.

[HH89] P. Horowitz and W. Hill. *The Art of Electronics*. Cambridge University Press, New York, NY, USA, 1989.

[HKV06] S. Habermann, R. Kothe, and H.T. Vierhaus. Built-in self repair by reconfiguration of fpgas. In *IOLTS 2006: Proceedings of the 12th IEEE International Symposium on On-Line Testing*, pages 187–188, 2006.

[HM01] Wei-Je Huang and Edward J. McCluskey. Column-based precompiled configuration techniques for fpga. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 137–146, 2001.

[HMR+00] Jerome C. Huck, Dale Morris, Jonathan Ross, Allan D. Knies, Hans Mulder, and Rumi Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, fourth edition, 2007.

[HX10a] Lin Huang and Qiang Xu. Agesim: a simulation framework for evaluating the lifetime reliability of processor-based socs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 51–56, 2010.

[HX10b] Lin Huang and Qiang Xu. Economic analysis of testing homogeneous manycore chips. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(8):1257–1270, 2010.

[Ise06] Rolf Isermann. *Fault-Diagnosis Systems - An Introduction from Fault Detection to Fault Tolerance.* Springer, 2006.

[ITR07] ITRS. International technology roadmap for semiconductors 2007 - design. Technical report, 2007. www.itrs.net.

[ITR10] ITRS. International technology roadmap for semiconductors 2010 - executive summary. Technical report, 2010. www.itrs.net.

[ITR11] ITRS. International technology roadmap for semiconductors 2011 - executive summary. Technical report, 2011. www.itrs.net.

[JDPK09] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-aware optimization targeting multithreaded stream computing. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 62–70, 2009.

[Jed06] Failure mechanisms and models for semiconductor devices. Technical Report JEP122C, JEDEC Solid State Technology Association, 03 2006.

[Jun11] Thorsten Jungeblut. *Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren*. PhD thesis, Universität Bielefeld, 2011.

[KK07] Israel Koren and C. Mani Krishna. *Fault-tolerant systems*. Elsevier / Morgan Kaufmann Publishers, 2007.

[KKC98] Way Kuo, Taeho Kim, and Wei-Ting Kary Chien. *Reliability, Yield, and Stress Burn-in: A Unified Approach for Microelectronics Systems Manufacturing and Software Development*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[KKS06] Sanjay V. Kumar, Chris H. Kim, and Sachin S. Sapatnekar. An analytical model for negative bias temperature instability. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 493–496, 2006.

[KKS07] Sanjay V. Kumar, Chris H. Kim, and Sachin S. Sapatnekar. Nbti-aware synthesis of digital circuits. In *Proceedings of the 44th annual Design Automation Conference*, pages 370–375, 2007.

[Kon07] Alexei Konnov. *Zuverlaessigkeitsberechnung und vorbeugende Wartung von komplexen technischen Systemen mittels modifizierter Markov-Methode*. PhD thesis, Universitaet Karlsruhe, 2007.

[KS90] I. Koren and A.D. Singh. Fault tolerance in vlsi circuits. *Computer*, 23(7):73–83, 1990.

[KVC+06] R. Kothe, H. T. Vierhaus, T. Coym, W. Vermeiren, and B. Straube. Embedded self repair by transistor and gate level reconfiguration.

In *IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 208–213, 2006.

[KZK+98] Ilyoung Kim, Y. Zorian, G. Komoriya, H. Pham, F.P. Higgins, and J.L. Lewandowski. Built in self repair for embedded high density sram. In *Test Conference, 1998. Proceedings., International*, pages 1112–1119, 1998.

[Lal01] Parag K. Lala. *Self-Checking and Fault-Tolerant Digital Design.* Elsevier / Morgan Kaufmann Publishers, 2001.

[LJ05] J. Lienig and G. Jerke. Electromigration-aware physical design of integrated circuits. In *Proc. 18th International Conference on VLSI Design*, pages 77–82, 2005.

[LK02] P.K. Lala and B.K. Kumar. An architecture for self-healing digital systems. In *Proc. Eighth IEEE International On-Line Testing Workshop*, pages 3–7, 2002.

[LKK+94] R. Leveugle, Z. Koren, I. Koren, G. Saucier, and N. Wehn. The hyeti defect tolerant microprocessor: a practical experiment and its cost-effectiveness analysis. *Computers, IEEE Transactions on*, 43(12):1398–1406, 1994.

[LLY+01] Wei Li, Qiang Li, Jiann-Shiun Yuan, J. McConkey, Yuan Chen, S. Chetlur, J. Zhou, and A.S. Oates. Hot-carrier-induced circuit degradation for 0.18 mu;m cmos technology. In *Quality Electronic Design, 2001 International Symposium on*, pages 284–289, 2001.

[LYHW03] Jin-Fu Li, Jen-Chieh Yeh, Rei-Fu Huang, and Cheng-Wen Wu. A built-in self-repair scheme for semiconductor memories with 2-d redundancy. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 393–402, 2003.

[Man08] W.R. Mann. Wafer test methods to improve semiconductor die reliability. *IEEE Design & Test of Computers*, 25(6):528–537, 2008.

[MDCS98] Ragavan Manian, Joanne Bechta Dugan, David Coppit, and Kevin J. Sullivan. Combining various solution techniques for dynamic

fault tree analysis of computer systems. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, pages 21–28, 1998.

[MHS+04] Subhasish Mitra, Wei-Je Huang, Nirmal R. Saxena, Shu-Yi Yu, and Edward J. McCluskey. Reconfigurable architecture for autonomous self-repair. *IEEE Design & Test of Computers*, 21(3):228–240, 2004.

[MLN+06] Madhu Mutyam, Feihui Li, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. Compiler-directed thermal management for vliw functional units. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 163–172, 2006.

[Moo98] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[MPKSZ05] Erik Jan Marinissen, Betty Prince, Doris Keitel-Schulz, and Yervant Zorian. Challenges in embedded memory design and test. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, pages 722–727, 2005.

[NAA04] M. Nicolaidis, N. Achouri, and L. Anghel. A diversified memory built-in self-repair approach for nanotechnologies. In *Proc. 22nd IEEE VLSI Test Symposium*, pages 313–318, 2004.

[NHS01] Tat Ngai, Chen He, and E.E.Jr. Swartzlander. Enhanced concurrent error correcting arithmetic unit design using alternating logic. In *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 78–83, 2001.

[Nic06] M. Nicolaidis. A low-cost single-event latchup mitigation scheme. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, pages 5 pp.–, 2006.

[Nic07] Paul E. Nicollian. *Physics of Trap Generation and Electrical Breakdown in Ultra-thin SiO2 and SiON Gate Dielectric Materials*. PhD thesis, University of Twente, August 2007.

[NK03] A.K. Nieuwland and R.P. Kleihorst. The positive effect on ic yield of embedded fault tolerance for seus. In *Proc. 9th IEEE On-Line Testing Symposium IOLTS 2003*, pages 75–79, 2003.

[OMM02] N. Oh, S. Mitra, and E.J. McCluskey. Ed4i: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180 –199, feb 2002.

[Pag03] Manish P. Pagey. *Hot-Carrier Reliability Simulation in aggressively scaled MOS Transistors*. PhD thesis, Graduate School of Vanderbilt University, 2003.

[PASM08] K. Pekmestzi, N. Axelos, I. Sideris, and N. Moshopoulos. A bisr architecture for embedded memories. In *Proc. 14th IEEE International On-Line Testing Symposium IOLTS '08*, pages 149–154, 2008.

[PGH+06] Mihalis Psarakis, Dimitris Gizopoulos, Miltiadis Hatzimihail, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. Systematic software-based self-test for pipelined processors. In *Proceedings of the 43rd annual Design Automation Conference*, pages 393–398, 2006.

[PGSR10] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27:4–19, 2010.

[PKK09] Abhisek Pan, Omer Khan, and Sandip Kundu. Improving yield and reliability of chip multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 490–495, 2009.

[PML02] Praveen Parvathala, Kaila Maneparambil, and William Lindsay. Frits: A microprocessor functional bist method. In *Proceedings of the 2002 IEEE International Test Conference*, pages 590–, 2002.

[Pra96] Dhiraj K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[PTB+10] Sokrates T. Pantelides, L. Tsetseris, M.J. Beck, S.N. Rashkeev, G. Hadjisavvas, I.G. Batyrev, B.R. Tuttle, A.G. Marinopoulos, X.J. Zhou, D.M. Fleetwood, and R.D. Schrimpf. Performance, reliability, radiation effects, and aging issues in microelectronics - from atomic-scale physics to engineering-level modeling. *Solid-State Electronics*, 54(9):841 – 848, 2010.

[PYF+00] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pages 90–95. 2000.

[PZK07] A. Pillai, Wei Zhang, and D. Kagaris. Detecting vliw hard errors cost-effectively through a software-based approach. In *Proc. 21st International Conference on Advanced Information Networking and Applications Workshops AINAW '07*, pages 811–815, 2007.

[RH03] M. Rausand and A. Høyland. *System Reliability Theory: Models, Statistical Methods, and Applications*. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. Wiley, 2003.

[RK11] R. Rodrigues and S. Kundu. On graceful degradation of chip multiprocessors in presence of faults via flexible pooling of critical execution units. In *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pages 67–72, 2011.

[RMC06] Kaushik Roy, T.M. Mak, and Kwang-Ting (Tim) Cheng. Test consideration for nanometer-scale cmos circuits. *IEEE Design & Test of Computers*, 23:128–136, 2006.

[RS08] Bogdan F. Romanescu and Daniel J. Sorin. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 43–51, 2008.

[SA98] Jian Shen and Jacob A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings of the 1998 IEEE International Test Conference*, pages 990–999, 1998.

[SABR04a] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. 31st Annual International Symposium on Computer Architecture*, pages 276–287, 2004.

[SABR04b] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. International Conference on Dependable Systems and Networks*, pages 177–186, 2004.

[SB03] Dieter K Schroder and Jeff A Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics*, 94:1, 2003.

[SBK06] Dennis Sylvester, David Blaauw, and Eric Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design & Test of Computers*, 23(6):484–490, 2006.

[Sch06] Mario Schölzel. *Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren*. PhD thesis, BTU Cottbus, 2006.

[Sch10] Mario Schölzel. Software-based self-repair of statically scheduled superscalar data paths. In *DDECS*, pages 66–71, 2010.

[Sch11] Mario Schölzel. Fine-grained software-based self-repair of vliw processors. In *Proceedings of the 2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pages 41–49, 2011.

[SCP+06] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. *SIGPLAN Not.*, 41(11):73–82, October 2006.

[Shi08] Jeonghee Shin. *Lifetime reliability studies for microprocessor chip architecture*. PhD thesis, UNIVERSITY OF SOUTHERN CALIFORNIA University of Southern California, March 2008.

[Sie91] D. P. Siewiorek. Architecture of fault-tolerant computers: An historical perspective. In *Proceedings of the IEEE*, pages 1710–1734, December 1991.

[Sin02] B.P. Singh. *Advanced Microprocessor and Microcontrollers*. New Age International, 2002.

[SKMB03] Premkishore Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. 21st International Conference on Computer Design*, pages 481–488, 2003.

[SKSH02] Jaume Segura, Ali Keshavarzi, Jerry Soden, and Charles Hawkins. Parametric failures in cmos ics : A defect-based analysis. In *Proceedings of the 2002 IEEE International Test Conference*, pages 90–, 2002.

[SLR+08] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, 2008.

[SPP01] V. Schober, S. Paul, and O. Picot. Memory built-in self-repair using redundant words. In *Proc. International Test Conference*, pages 995–1001, 2001.

[SPR04] Naran Sirisantana, Bipul C. Paul, and Kaushik Roy. Enhancing yield at the end of the technology roadmap. *IEEE Design & Test of Computers*, 21(6):563–571, 2004.

[SR03] J. Stinson and S. Rusu. A 1.5 ghz third generation itanium 2 processor. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 252–492 vol.1, 2003.

[Sri06] Jayanth Srinivasan. *Lifetime reliability aware microprocessors*. PhD thesis, University of Illinois, 2006.

[SS99] Ranganathan Sankaralingam and Wade Schwartzkopf. Evaluation of the tms320c6x c compiler and assembly optimizer, 1999.

[Sta02] J. H. Stathis. Reliability limits for the gate insulator in cmos technology. *IBM J. Res. Dev.*, 46:265–286, March 2002.

[Ste00] Andreas Steininger. Testing and built-in self-test - a survey. *Journal of Systems Architecture*, 46(9):721–747, 2000.

[SYW05] Chin-Lung Su, Yi-Ting Yeh, and Cheng-Wen Wu. An integrated ecc and redundancy repair scheme for memory reliability enhancement. In *Proc. 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems DFT 2005*, pages 81–89, 2005.

[SZBP08] Jeonghee Shin, V. Zyuban, P. Bose, and T.M. Pinkston. A proactive wearout recovery approach for exploiting microarchitectural redundancy to extend cache sram lifetime. In *Proc. 35th International Symposium on Computer Architecture ISCA '08*, pages 353–362, 2008.

[SZH+07] Jeonghee Shin, V. Zyuban, Zhigang Hu, J.A. Rivers, and P. Bose. A framework for architecture-level lifetime reliability modeling. In *Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN '07*, pages 534–543, 2007.

[TA80] S.M. Thatte and J.A. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, 29:429–441, 1980.

[TIC+05] Darshan D. Thaker, Francois Impens, Isaac L. Chuang, Rajeevan Amirtharajah, and Frederic T. Chong. Recursive tmr: Scaling fault tolerance in the nanoscale era. *IEEE Design & Test of Computers*, 22(4):298–305, 2005.

[TWH+07] Tsu-Wei Tseng, Chun-Hsien Wu, Yu-Jen Huang, Jin-Fu Li, A. Pao, K. Chiu, and E. Chen. A built-in self-repair scheme for multiport

rams. In *Proc. 25th IEEE VLSI Test Symposium*, pages 355–360, 2007.

[vdWVD+05] J.-W. van de Waerdt, S. Vassiliadis, Sanjeev Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, P. Rodriguez, and H. van Antwerpen. The tm3270 media-processor. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12 pp.–342, 2005.

[vN56] J. von Neumann. *Automata Studies*, chapter Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components. Princeton Univ. Press, 1956.

[VS07] Mojtaba Valinataj and Saeed Safari. Fault tolerant arithmetic operations with multiple error detection and correction. In *Proc. 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems DFT '07*, pages 188–196, 2007.

[VVB+09] J. Vial, A. Virazel, A. Bosio, L. Dilillo, P. Girard, C. Landrault, and S. Pravossoudovitch. Using tmr architectures for soc yield improvement. In *Proc. First International Conference on Advances in System Testing and Validation Lifecycle VALID '09*, pages 155–160, 2009.

[WA01] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Proc. International Conference on Dependable Systems and Networks DSN 2001*, pages 411–420, 2001.

[Whi10] Mark White. *Scaled CMOS technology reliability users guide.* Pasadena, CA : Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2010., January 2010.

[WWW06] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon).* Morgan Kaufmann Publishers Inc., 2006.

[YHC11] Hao-I Yang, Wei Hwang, and Ching-Te Chuang. Impacts of gate-oxide breakdown on power-gated sram. *Microelectronics Journal*, 42(1):101 – 112, 2011.

[ZGVM04] Y. Zorian, D. Gizopoulos, C. Vandenberg, and P. Magarshack. Guest editors' introduction: Design for yield and reliability. *IEEE Design & Test of Computers*, 21(3):177–182, 2004.

Beteiligte Publikationen

[1] C. Gleichner, T. Koal, and H.T. Vierhaus. Effective logic self repair based on extracted logic clusters. In *Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference Proceedings (SPA), 2010*, pages 1–6, 2010.

[2] T. Koal, C. Galke, and H.T. Vierhaus. Software-based self test for embedded processors enhanced by structural information. *Forum der Forschung*, 20:79–84, 2007.

[3] T. Koal, D. Scheit, M. Schölzel, and H.T. Vierhaus. On the feasibility of built-in self repair for logic circuits. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*, pages 316–324, 2011.

[4] T. Koal, D. Scheit, and H.T. Vierhaus. A concept for logic self repair. In *Proc. 12th Euromicro Conference on Digital System Design (DSD)*, pages 621–624, 2009.

[5] T. Koal, D. Scheit, and H.T. Vierhaus. Reliability estimation process. In *Digital Systems Design, Euromicro Symposium on*, pages 221–224, 2009.

[6] T. Koal, D. Scheit, and H.T. Vierhaus. A scheme of logic self repair including local interconnects. In *Proc. 12th Euromicro Conference on Digital System Design (DSD)*, pages 8–11, 2009.

[7] T. Koal, D. Scheit, and H.T. Vierhaus. Selbstreparatur durch Regularisierung von Logik-Schaltungen. In *Proc. 3. GMM/GI/ITG Fachtagung Zuverlässigkeit und Entwurf (ZuE)*, 2009.

[8] T. Koal, M. Schölzel, and H.T. Vierhaus. Dependability and life time enhancements for nano-electronic systems. In *Signal Processing Algorithms, Ar-*

chitectures, Arrangements, and Applications Conference Proceedings (SPA), 2011, pages 1–6, 2011.

[9] T. Koal, M. Ulbricht, P. Engelke, and H.T. Vierhaus. On the feasibility of combining on-line-test and self repair for logic circuits. In *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 187–192, 2013.

[10] T. Koal, M. Ulbricht, and H.T. Vierhaus. Combining on-line fault detection and logic self repair. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pages 288–293, 2012.

[11] T. Koal, M. Ulbricht, and H.T. Vierhaus. Virtual tmr schemes combining fault tolerance and self repair. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 235–242, 2013.

[12] T. Koal and H.T. Vierhaus. Funktionaler Selbsttest für eingebettete Prozessoren auf der Basis struktureller Information. In *20. Workshop für Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ)*, 2008.

[13] T. Koal and H.T. Vierhaus. A software-based self-test and hardware reconfiguration solution for vliw processors. In *IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 40–43, 2010.

[14] T. Koal and H.T. Vierhaus. Optimal spare utilization for reliability and mean lifetime improvement of logic built-in self-repair. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 219–224, 2011.

[15] M. Schölzel, T. Koal, S. Röder, and H.T. Vierhaus. Towards an automatic generation of diagnostic in-field sbst for processor components. In *Test Workshop (LATW), 2013 14th Latin American*, pages 1–6, 2013.

[16] M. Scholzel, T. Koal, and H.T. Vierhaus. An adaptive self-test routine for in-field diagnosis of permanent faults in simple risc cores. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pages 312 –317, April 2012.

[17] M. Ulbricht, T. Koal, and H.T. Vierhaus. Activity migration in m-of-n-systems by means of load-balancing. In *Proceedings of the 15th Euromicro Conference on Digital Systems Design (DSD)*, pages 258–263, 2012.

[18] M. Ulbricht, M. Schölzel, T. Koal, and H.T. Vierhaus. A new hierarchical built-in self-test with on-chip diagnosis for vliw processors. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 143 –146, April 2011.